Universitat de Lleida

TREBALL FINAL DE MÀSTER

Estudiant: **Roger Truchero Visa**

Titulació: Màster en Enginyeria Informà

Títol de Treball Final de Màster: Efficient smishing detector using Machine Learning techniques

Director/a: Francesc Giné de Sola

Presentació

Mes: Juliol

Any: 2021

Preamble

The main reason for studying and carrying out this project is the interest in the burning problem, in the company where I currently work. To be able to protect our users from text messages with malicious content we need to detect it and filter. Every day we receive millions of text messages and not always the non-legitimate messages are filtered effectively.

The general purpose of the project is to develop a tool that from an SMS is capable of detecting in real time whether it contains phishing or not, to finally discard it. To perform this task, an analysis of the state of the art will be carried out to know the current systems with greater success and efficiency scores, to later implement and integrate the system with the company's SMS sending service and filter the messages they contains phising in real time.

From the beginning, it seemed to me an interesting project where many knowledges and strategies converged and in which I could acquire new ones. As well as a great opportunity to make a tool that gives an extra layer of protection to our users and enrich the other involved company services.

Acknowledgments

I have enjoyed practically the whole project, since I am not a lover of documentation. For me this work has been more than just a project to deliver. I have seen it as a point of learning and self-improvement which has led me to research and find my own way to achieve my goals. It would be selfish to say that all the work has been mine, so I would like to thank my family and friends for the indirect support they have had on the work giving me encouragement and making my head partially disconnect from *the magical world of bits*.

Not less important, I would also like to thank my tutors of the project, Sisco Giné and David Tàpia, for the confidence that they deposited in me for the realization of the project apart from contributing with new ideas and points of improvement. Also to Lleida.net for letting me present such a fun and useful project for their customers.

People worry that computers will get too smart
and take over the world, but the real problem
is that they're too stupid and they've already taken over the world.

Pedro Domingos.

Contents

List of Figures

List of Tables

List of Acronyms and Abbreviations

API	**A**pplication **P**rogramming **I**nterface.
APK	Android **A**pplication **P**ac**k**age.
BR	**B**ussiness **R**equirements.
BSD	**B**erkeley **S**oftware **D**istribution.
BYOD	**B**ring **Y**our **O**wn **D**evice.
CEO	**C**hief **E**xecutive **O**fficer.
CPU	**C**entral **P**rocessing **U**nit.
CURL	**c**lient **URL**.
DH	**D**iffie-**H**ellman.
DTs	**D**ecision **T**ree**s**.
FN	**F**alse **N**egative.
FP	**F**alse **P**ositive.
FR	**F**unctional **R**equirements.
GIL	**G**lobal **I**nterpreter **L**ock.
HTTP	**H**yper**T**ext **T**ransfer **P**rotocol.
HTTPS	**H**yper**T**ext **T**ransfer **P**rotocol **S**ecure.
ICX	**I**nter**C**onnection e**X**change.
IMAP	**I**nternet **M**essage **A**ccess **P**rotocol.
IP	**I**nternet **P**rotocol.
IREP	**I**ncremental **R**educed **E**rror **P**runing.
IT	**I**nformation **T**echnology.
JSON	**J**ava**S**cript **O**bject **N**otation.
MIME	**M**ultipurpose **I**nternet **M**ail **E**xtensions.
NFR	**N**on-**F**unctional **R**equirements.
NLP	**N**atural **L**anguage **P**rocessing.
NLU	**N**atural **L**anguage **U**nderstanding.
PID	**P**rocess **ID**entificator.
PIP	**P**ip **I**nstalls **P**ackages.
POP3	**P**ost **O**ffice **P**rotocol.
QA	**Q**uality **A**ssurance.
RAM	**R**andom **A**ccess **M**emory.
RE	**R**equirements **E**ngineering.
REST	**RE**presentational **S**tate **T**ransfer.
RIPPER	**R**epeated **I**ncremental **P**runing to **P**roduce **E**rror **R**eduction.

RSA	**R**ivest **S**hamir & **A**dleman.
SAL	**S**elf **A**nswering **L**ink.
SMS	**S**hort **M**essage **S**ervice.
SMTP	**S**imple **M**ail **T**ransfer **P**rotocol.
SQL	**S**tructured **Q**uery **L**anguage.
SSH	**S**ecure **SH**ell.
SSL	**S**ecure **S**ockets **L**ayer.
SVN	**S**ub**v**ersio**n**.
TFIDF	**T**erm **F**requency **I**nverse **D**ocument **F**requency.
TFM	**T**rabajo **F**inal de **M**aster.
TLS	**T**ransport **L**ayer **S**ecurity.
TN	**T**rue **N**egative.
TNR	**T**rue **N**egative **R**ate.
TP	**T**rue **P**ositive.
TPR	**T**rue **P**ositive **R**ate.
UR	**U**ser **R**equirements.
URI	**U**niform **R**esource **I**dentifier.
URL	**U**niform **R**esource **L**ocator.
VSCode	**V**isual **S**tudio **Code**.
WBS	**W**ork **B**reakdown **S**tructure.
WSGI	**W**eb **S**erver **G**ateway **I**nterface.

1. Introduction

In order to face our work, we need to introduce the main concepts where our work will focus, the smishing.
Smishing is a word that is made up of **S**hort **M**essage **S**ervice (SMS) and **phishing**[1]. Phishing or spam attacks are very common in emails but SMS continue to be considered by users as legitimate shipments, since they are usually used for personal communication, bank notifications, air lines or single-use codes to validate operations or accesses. It is for this reason that cybercriminals are incorporating them into their repertoire of attack techniques to access user data.

This is what is called **smishing**, a social engineering technique by which cybercriminals attack in a massive way and target many users by sending an SMS pretending to be a legitimate recipient that can be, for example, a bank, a social network, a public institution or a widely used application. The objective of these attacks is to steal private information or make financial charges on the victim's accounts, prompting the user to access a fake web link or to enter their credentials to confirm their account. What makes this type of cyber attack so dangerous is the lack of habit and prevention on the part of users.

Attackers use SMS because it is cheap, it is easy to obtain phone listings, and it can be programmed to send in bulk. In addition, they have a point in their favour over fraudulent emails, how little used we are to them. It is for this reason that multiple solutions have been considered to mitigate and eliminate messages that are considered as smishing.

Currently there are multiple smishing detection and filtering systems but it is always recommended to follow a series of steps to avoid possible deception such as:

- Be wary of unknown senders
- Never provide information
- Do not click on links
- Block messages that you consider spam
- Personalize security options to use double verification systems
- Verify the sender

[1] The fraudulent practice of sending emails purporting to be from reputable companies in order to induce individuals to reveal personal information, such as passwords and credit card numbers.

Millions of SMS's from multiple countries around the world are received every day at the company I work for, **Lleida.net**. **Lleida.net** is the first Certification Operator, a reference company in the field of certified communications and telecommunications whose mission is to bring security, trust, efficiency and profitability to the electronic communications of companies, public administrations and individuals, directly influencing the improvement of their results.

Many of the received messages are unfortunately fraudulent and have an illegitimate background, the main objective is to filter all these messages more effectively with the highest rate of successness, both for false-positives and false-negatives, and to guarantee a greater security for our customers. This is a motivating project given that it is an issue that currently affects millions of people, to be more precise, anyone who has a mobile device with SMS services. Also exists the purpose of aporting new strategies and approaches to improve the current systems or simply offer another point of view.

1.1. Motivation

The main interest of this project is to provide an extra layer of security to our customers by being able to effectively detect messages that contain smishing and those that do not. In many cases, people with little knowledge of current technologies and their great diversification are not aware of the risks involved in opening an SMS link without first carefully checking its content, or sending personal information to an unfamiliar email. That is why one of the most effective tasks to prevent this type of deception is to carry out campaigns to raise awareness of the risks and consequences of this type of action, but even so it is not enough. Therefore, in order to offer the best possible protection, this project aims to detect as many messages with malicious content as possible and to effectively cut off the traffic coming from that source.

1.2. State of art

In recent years, researchers are focusing more on **smishing** because of its popularity in mobile attacks. Some researchers have proposed models for detecting messages containing smishing and many have analysed strategies and approaches for their detection. This section will focus on introducing above techniques that are being used to mitigate smishing attacks. As we introduced previously, smishing is a category of **phishing** and therefore, similar techniques are used for its detection.

We will start with the study carried out by Mishra et al. [9], authors of the "Smishing Detector" system. This system is mainly based on four components named:

1. SMS Content Analyzer
2. URL Filter
3. Source Code Analyzer

4. APK Download Detector

SMS Content Analyzer verifies the presence of URL, **S**elf **A**nswering **L**ink (SAL), phone number and email id in the SMS. Messages containing email id and phone number are processed for blacklist check. The messages are forwarded for text preprocessing to categorize the message on the basis of the keywords present on it. Keywords contained in the message are classified using **TfidfVectorizer**[2] and **Naive Bayes**[3] classifier.

URL Filter first converts the short URL to long URL then, it looks for the URL in the blacklist. URL found in the blacklist is categorized as Smishing. This module also verifies four features of the url, namely, the age of domain, presence of *@tag*, presence of hyphen and number of dots present in the URL to check the authenticity of it. If the threshold of the above features is greater than or equal three, the categorize the message as Smishing, otherwise pass it to the Source Code Analyzer.

Source Code Analyzer verifies the presence of any form tag in the source code. If form tag is present in the source code, Source Code Analyzer compares the domain of the request URL in source code with the domain of the actual URL invoked. If the domain is different, the message is classified as Smishing, otherwise URL is transferred to Android **A**pplication **P**ac**k**age (APK) Download Detector.

APK Download Detector checks for any file downloading while invoking the URL. This checking is carried without visiting the website. The base name of actual URL is extracted to assess whether it contains .apk extension as part of the base name. It also checks whether the file is downloaded with user permission or not, because some files are downloaded after re-direction of web pages on phishing websites.

The final prototype experimented shown an **accuracy of 96.29%**.

Another study proposed by Jain et al. [7] suggested a rule-based method for identifying Smishing messages. They have identified **nine** rules to filter smishing messages from legitimate messages then use the classification algorithm namely **Decision Tree, Repeated Incremental Pruning to Produce Error Reduction (RIPPER)[4] and PRISM[5]** to apply it.

[2]Transforms text to feature vectors that can be used as input to estimator.

[3]A naive Bayes classifier is an algorithm that uses *Bayes* theorem to classify objects. Naive Bayes classifiers assume strong, or naive, independence between attributes of data points. Popular uses of naive Bayes classifiers include spam filters, text analysis and medical diagnosis.

[4]The RIPPER algorithm was introduced by W. Cohen in 1995, which improved upon **I**ncremental **R**educed **E**rror **P**runing (IREP) to generate rules that match or exceed the performance of decision trees. Having evolved from several iterations of the rule learning algorithm, the RIPPER algorithm can be understood in a three-step process: Grow, Prune and Optimize.

[5]Prism algorithm is a rule based algorithm that induces modular rules using **separate and conquer** approach. Training data set may have both or either categorical or numerical attribute.

These rules are the following:

- **Rule 1**: **If** URL present in the message, **then** it is probably a smishing message. A URL analyzer checks for the presence of URL in the text message since attackers can trick users by sending a URL link in the text message which when opened can direct the user to either a malicious login page or can download a malware in the user's mobile phone.
- **Rule 2**: **If** the message contains any mathematical symbol like +, -, <, >, /, etc., **then** it is a probably a smishing message.
- **Rule 3**: **If** message contains any currency sign like $, £, etc., **then** it is probably a smishing message. For example, specific symbol $ is being used to represent money in the fake award messages.
- **Rule 4**: **If** a mobile number present in the message, **then** it is probably a smishing message. The attacker asks the users to send the user's details, bank details, on given number.
- **Rule 5**: The presence of suspicious keywords like, *free, accident, awards, dating, won, service, lottery, mins, visit, delivery, cash, claim, prize, delivery, etc.* are considered as smishing keywords. **If** any of the suspicious keyword present in the message, **then** it is a presumably a smishing message.
- **Rule 6**: **If** message length is greater than 150 character, **then** it is potentially a smishing message. This length including space, symbols, special characters, smileys, etc.
- **Rule 7**: **If** message is the SAL type, **then** it is a likely a smishing message. The presence of SAL SMS asks the user to subscribe or unsubscribe any service.
- **Rule 8**: **If** message contains **visual morphemes**[6], **then** it is probably a smishing message.
- **Rule 9**: **If** message contains the email address, **then** it is likely a smishing message. The attacker also uses the email address in the message to get the personal information on the desired source.

The results in Table 1.1 shows that the RIPPER algorithm outperformed *Decision Tree* and *PRISM* in terms of **T**rue **P**ositive **R**ate (TPR). For the **T**rue **N**egative **R**ate (TNR) all algorithms overcome the **99%**. So if we estimate the global accuracy taking into account the TPR and TNR we obtain an accuracy of **95.02%**.

A research work proposed by Sonowal et al. [13] shows a model named **SmiDCA** for detecting smishing message using also a machine learning approach as is shown in Fig.1.1. This model initially investigates the data and extracts the distinct features. Afterwards, the model ranks all of those features by exploiting correlation algorithm and generates the subset

[6] A morpheme is the smallest meaningful unit in a language. Visual Morphemes are numerals and other signs used in writing text messages, emails, etc.

Algorithm	True Positive Rate	True Negative Rate	False Positive Rate	False Negative Rate
Decision tree	90.88%	99.17%	0.86%	9.12%
PRISM	72.65%	99.93%	0.07%	27.35%
RIPPER	92.92%	99.01%	0.99%	7.18%

Table 1.1.: Results for Rule-Based model

by adding of high ranked features one by one and sending them to the machine learning algorithm.

The learning algorithm evaluates the accuracy and verifies whether the accuracy is better than the previous accuracy or not. **If** the accuracy is increased, **then** adds the next high ranked features to the subset, **otherwise** terminates the process.

This model shown an experimental evaluation **accuracy of 96.40%** using **Random For-**

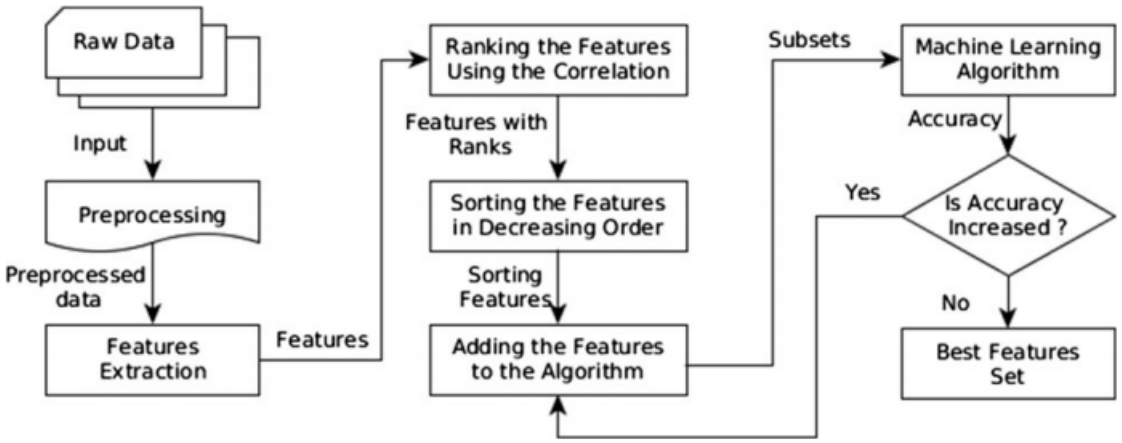

Figure 1.1.: SmiDCA scheme

est[7] classifier.

At first glance we can see that the 3 proposed models have a high degree of accuracy over the theory, all of them above 95%. While it is true that each of the proposed solutions are very different from each other, the one who obtains the highest accuracy is **SmiDCA** with 96.40%, followed by **SmishingDetector** with 96.29% and finally the rule-based solution with 95.02%. In the next chapter we will perform a more detailed and precise analysis to investigate the different strategies and evaluate what could be our starting point.

[7]Random Forest is a robust machine learning algorithm that can be used for a variety of tasks including regression and classification. It is an ensemble method, meaning that a random forest model is made up of a large number of small decision trees, called estimators, which each produce their own predictions. The random forest model combines the predictions of the estimators to produce a more accurate prediction.

1.3. Research questions

The main research question to be answered is:

How can we design a system able to detect smishing SMS effectively and efficiently to protect our customers?

To answer this question, several sub questions have to be answered previosuly:

1. *What is smishing and what types of messages are commonly used?*
2. *What measures exists to combat smishing?*
3. *What smishing detection systems currently exist and what do they consist of?*
4. *Which requirements should our system have to have in order to be implemented in our company?*
5. *What validation modules will our system have?*
6. *How will we evaluate the accuracy and performance of our system?*
7. *What tools will we use to optimise our system?*
8. *How will we deploy our system?*

1.4. Methodology

In order to answer each of the sub-questions mentioned above, we will have to carry out a detailed preliminary analysis to guarantee the highest quality on the final system.

The **first** and **second** questions aim to obtain basic knowledge about the main topic of the project, smishing, in a simple and informative way to make a first approach to the context to be treated. The **third** question is aimed at carrying out a search of current works related to our topic and analysing them in order to extract the most fundamental aspects, already thinking about the design of our final system. In the **fourth** question, we will focus on holding meetings to narrow down the requirements of our work colleagues from the **Core department** so that they can express their points of view and needs in order to be able to integrate our system quickly and easily. In order to answer the **fifth** question, the previous questions will have to be answered, as they have a direct correlation with the analysis before the system design phase. This point will be based on designing the system in a generalist way and carefully choosing each decision in order to guarantee the highest possible degree of quality. The **sixth** and **seventh** questions aim to evaluate and optimise our implementation, since performance is one of the main requirements. Finally, the **eighth** question will be based on analysing and deploying, conditional on the requirements, our system as a validation application so that our colleagues can make use of the functionalities it provides.

1.5. Scheduling and Budget

In this section, we will plan the main activities to be performed during our project. Also, we will describe the planning of these activities during the development of the whole project.

1.5.1. Scheduling

To summarize and estimate the development time, we'll detail all tasks grouping by "task packages". In table 1.2 we can see this in more detail.

From the "task packages" estimation Table 1.2 we'll develop a Gantt chart for each activity. To reduce the space of the Gantt diagram we will estimate the time consumed by each task in weeks. Since the project is carried out in conjunction with the company's tasks, only 4 hours of work per day are available. This available time may vary depending on the other workloads, but we will try to make the estimate as accurate as possible. The Gantt diagram is shown in Figure 1.2.

1.5.2. Budget

This project targets our company **Lleida.net**. In the estimation, all costs associated with the tasks have been considered, as well as the software and hardware infrastructure costs required to deploy the application in a real environment.

The results shown in table 1.3 are done based on the price of a professional developer working on this project, we'll take an example of **30€** per hour. The software and hardware infrastructure costs required to bring the application to a real production environment have also been considered. These can be seen in Table 1.4.

Task	*Time Estimation (hours)*	*Time Estimation (days)*	*Time Estimation (weeks)*
1. Documentation	160	40	8
1.1. Thesis structure design	30	7,5	1,5
1.2. Chapters, sections and subsections redaction	120	30	6
1.3. Quality check	10	2,5	0,5
2. State of Art	30	7,5	1,5
3. Analysis	35	8,75	1,75
3.1. Smishing	3	0,75	0,15
3.2. Machine Learning smishing models	32	8	1,6
4. Requirements	15	3,75	0,75
4.1. Obtaining requirements	13	3,25	0,65
4.2. Classifying requirements	2	0,5	0,1
5. Design, Implementation and Evaluation	262	65,5	13,1
5.1. Architecture and flow design	40	10	2
5.2. Implementation	180	45	9
5.2.1. API	15	3,75	0,75
5.2.2. Core	135	33,75	6,75
5.2.2.1. Engine	30	7,5	1,5
5.2.2.2. SMSContentAnalyzer	15	3,75	0,75
5.2.2.3. Prediction model	40	10	2
5.2.2.3.1. English model	25	6,25	1,25
5.2.2.3.2. Spanish model	15	3,75	0,75
5.2.2.4. Blacklist	5	1,25	0,25
5.2.2.5. URLFilter	15	3,75	0,75
5.2.2.6. SourceCodeAnalyzer	15	3,75	0,75
5.2.2.7. APKDownloadDetector	15	3,75	0,75
5.2.3. Modules	30	7,5	1,5
5.3. Evaluation	42	10,5	2,1
5.3.1. Testing	15	3,75	0,75
5.3.2. Profiling	2	0,5	0,1
5.3.3. Optimization	25	6,25	1,25
6. Deployment	37	9,25	1,85
6.1. Design	10	2,5	0,5
6.2. Server and enviornment preparation	5	1,25	0,25
6.3. Deployment configurations	20	5	1
6.4. Testing	2	0,5	0,1
7. Results	12	3	0,6
TOTAL	***551***	***137,75***	***27,55***

Table 1.2.: Task packages estimation in hours, days and weeks

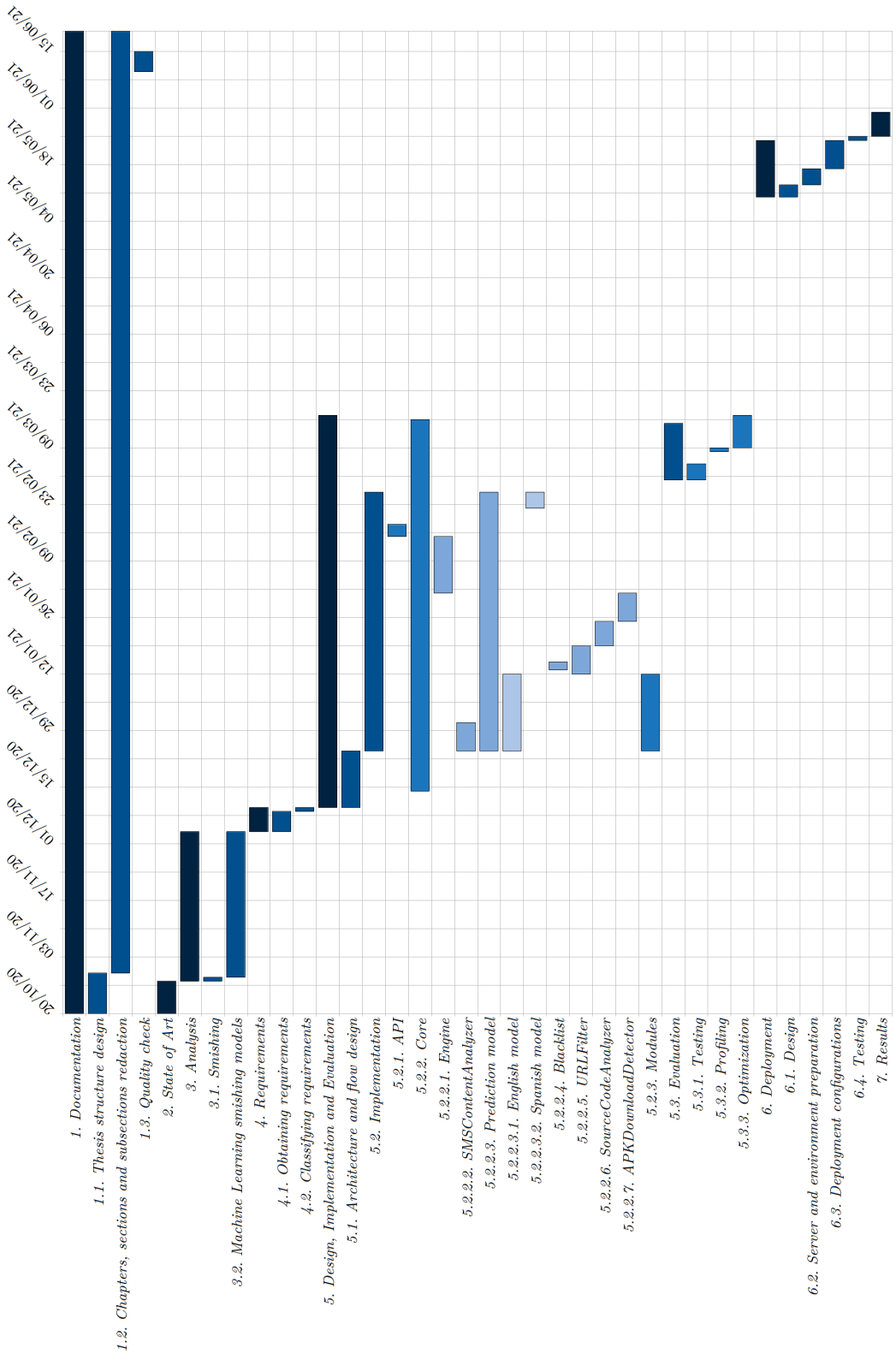

Figure 1.2.: Scheduled tasks Gantt

Task	*Time Estimation (hours)*	*Total Price (€)*
1. Documentation	160	4800
1.1. Thesis structure design	30	900
1.2. Chapters, sections and subsections redaction	120	3600
1.3. Quality check	10	300
2. State of Art	30	900
3. Analysis	35	1050
3.1. Smishing	3	90
3.2. Machine Learning smishing models	32	960
4. Requirements	15	450
4.1. Obtaining requirements	13	390
4.2. Classifying requirements	2	60
5. Design, Implementation and Evaluation	262	7860
5.1. Architecture and flow design	40	1200
5.2. Implementation	180	5400
5.2.1. API	15	450
5.2.2. Core	135	4050
5.2.2.1. Engine	30	900
5.2.2.2. SMSContentAnalyzer	15	450
5.2.2.3. Prediction model	40	1200
5.2.2.3.1. English model	25	750
5.2.2.3.2. Spanish model	15	450
5.2.2.4. Blacklist	5	150
5.2.2.5. URLFilter	15	450
5.2.2.6. SourceCodeAnalyzer	15	450
5.2.2.7. APKDownloadDetector	15	450
5.2.3. Modules	30	900
5.3. Evaluation	42	1260
5.3.1. Testing	15	450
5.3.2. Profiling	2	60
5.3.3. Optimization	25	750
6. Deployment	37	1110
6.1. Design	10	300
6.2. Server and enviornment preparation	5	150
6.3. Deployment configurations	20	600
6.4. Testing	2	60
7. Results	12	360
TOTAL	***551***	***16530***

Table 1.3.: Budget estimation based on the time consumed by all tasks

Product	*Price (€ / month)*
1. Hardware	77,16
Server hosting (8GB RAM + 4 CPU's + 50GB SSD)	67,17
MYSQL hosting (4GB RAM + 4 CPU's + 450GB SSD)	9,99
TOTAL	***77,16***

Table 1.4.: Budget estimation based on the price of each part every month

1.6. Thesis structure

In this first introductory **chapter** 1, the basic aspects such as the motivations and context of the project have been described, as well as the main search questions, the methodology for answering them and the scheduling with the estimation costs for the company. In **Chapter** 2 a more detailed analysis of the theory behind current smishing systems is provided. Then in **chapter** 3 we analyze and discuss about the requirements that the system will have, this phase is crucial in order to move on to the next phases associated with the design and implementation.

Chapter 4 describes the design, implementation and evaluation phases. We describe the design phase in detail: the aspects and decisions taken in the construction phase of the system, such as the structure and communication between classes or the database model. This phase focuses on an analysis with a less technical background as we are thinking abstractly in order not to close the doors to new technologies. For the implementation phase we will explain the processes and decisions taken in the implementation of the system as well as the functionalities they perform, approaching in a more technical way the concepts of development but in the clearest possible way. Once the system has been designed and implemented, we will focus on its evaluation, this implies testing the application to avoid failures and possible bugs. This is the previous step before we continue with the profiling and optimization step, using specific tools or libraries. When this is done it's time to improve the system in terms of performance and effectiveness.

Chapter 5 describes explicitly and in detail the phases of the deployment of our system as a web application, so that it can be used as an **A**pplication **P**rogramming **I**nterface (API) for its subsequent internal use, some points on the security of the system will also be discussed. Then in the **chapter** 6 we'll talk about the results and its interpretation. Finally, in **chapter** 7 we'll talk about the conclusions extracted from our project, the limitations and the future work.

2. Analysis

In this chapter we'll focus on describing and analyzing in detail the main concepts of our project, extracting a brew conclusion for each different section.

2.1. Smishing

In this section we will analyse in detail all the features related to smishing in order to have a more global view when making possible design decisions.

2.1.1. What is

As previously mentioned in chapter 1, the term smishing is equivalent to phishing but carried out via SMS. The purpose of smishing is to defraud and/or manipulate consumers or employees of an organisation.

Such messages usually contain a link which by clicking on redirects to a website that will attempt to get personal login details or other information about the user. Therefore, the cybercriminal's main objective is to obtain this data and then use it to access personal or work accounts, commit identity fraud or engage in some kind of malicious activity.

The definition of the term smishing sounds similar to phishing, that's because they are practically the same. Smishing is a sub-category of phishing, i.e. every smishing message is phishing, but not necessarily the other way around since phishing itself is a fairly broad term that describes fraudulent activities and cybercrimes against individuals and businesses alike, which can be carried out through multiple channels such as:

- **Email**: spear phishing, whaling, CEO fraud, payroll fraud, business email compromise ...
- **Phone calls**: vishing (voice phising)
- **Social media messages**: Linkedin, Facebook, Twitter, Instagram, ...

2.1.2. Scam types

There are countless types of deception, for now I will only mention a few of the ones used today, later we will see some real examples:

- Texts from banks, investment firms and other financial institutions stating there's an issue with your account.
- Messages promising free money, products or services.

- Text messages from companies and service providers stating that there's an issue and you need to update your payment account information.
- Messages from various "authorities" about COVID-19 contact tracing updates and various pandemic-related resources.

2.1.3. How it works

We can perform a smishing attack without getting into a lot of implementation complexity. You simply need to have a target (in some cases not necessary) and a few technologies at hand. Usually the more specific attacks involve the use of social engineering tactics.

Below, we will describe how a smishing attack unfolds:

1. **A cybercriminal sends an SMS text message from a spoofed number.** The content and number from which the text originates can make it appear to come from a legitimate source. In some cases when they already have more specific information beforehand they may even make the message even more specific, impersonating a service you are currently using (such as a streaming service or a bank).
2. **You receive the message on your phone expecting some kind of response.** The message may contain a tempting offer, or it could be something potentially worrying that prompts you to act on it (overdue debts or a repossession order).
3. **Action in response to the message.** This step is the determining factor in how things will play out. If you simply ignore the message or report it, that's basically the end of it, you will have managed to avoid the scam although you may still receive such messages. But if you click on the link, you will be redirected to a website that may look legitimate but is not. You will be asked to provide some information or download something (such as a device or browser update) before you can continue.
    a) You will be asked to provide information that you would not otherwise provide, this information may contain sensitive data such as a credit or debit card number, or even work login information.
    b) You will find yourself downloading something that contains malicious software, so by getting you to download the malware you will be giving them access to your device and your sensitive data.

2.1.4. Why is so dangerous

The report provided by Verizon's Mobile Security Index 2020 states that 17% of phishing occurs via text messages. In fact, many companies test their employees by performing these types of drills to see what percentage of people fall for the trap.

The following are the main reasons why smishing is such a dangerous practice and can be devastating for small to large businesses:

- **It's an easy and cost-effective attack vector.** As **Mørten Brogger**, CEO of the security platform *Wire* says,

> *"SMS phishing is heavily used by cybercriminals because it is one of the easiest, cheapest, and effective methods of cyberattacks. It works by exploiting human error, which is the greatest cybersecurity weakness of regular users and can never be truly prevented since everyone makes mistakes. Cybercriminals need only to send malicious links disguised in seemingly helpful messages with the name of familiar organizations attached to catch a few consumers off guard and gain access to their data. All it takes is a single mis-click for SMS phishing to succeed."*
> *— Morten Brøgger*

- **Your phone usually contains very sensitive data.** As **Pieter Vanlperen** says,

> *"SMS phishing is so dangerous because people today carry their entire lives on their mobile devices. There's an app for everything and we have the world at our fingertips. The problem lies in that mobile devices do not come with antivirus or antimalware programs, so if you click a link that is sent to you by SMS, you're opening the door for thieves. They can steal anything from your contact list to your phone permissions to your financial information, depending on what the author of the malware designed it to do. Since we keep everything on our mobile devices, attackers can gain access to our entire lives." — Pieter Vanlperen*

- **Smishing can get employees to give up sensitive information.** As **Reuben Yonatan**, founder and CEO of *GetVoIP*[1] says,

> *"Cybercriminals have been known to use smishing to blackmail an employee into revealing company secrets. An employee will get a text urging them to reveal x, y, z about the company or something horrible will happen imminently.*
>
> *Alternatively, the cybercriminal can make the employee reveal the secrets unwittingly. They can send a convincing text pretending to be the boss." — Reuben Yonatan*

- **Smishing enables cybercriminals to bypass traditional security mechanisms.** As **Mørten Brogger** says,

[1] https://getvoip.com/

> *"Businesses and organizations spend millions of dollars every year to secure sensitive data worth billions of dollars. SMS phishing can sneak malicious software through firewalls and secure networks by hitching a ride on an unsuspecting employee's phone. This is particularly concerning with the transition to remote work where employees are in a more relaxed environment at home without the secure networks, systems, and reminders to reinforce their adherence to proper procedures. These risks have loomed large this year as well known companies and people fell victim — the Twitter breach in July and successful hack of Jeff Bezos' phone (revealed in January), to name a few." — Morten Brøgger*

- **Smishing poses legal and financial ramifications for consumers & businesses.** As **Kaulum Ross**, founder of *What in Tech*[2] and Senior Technical Project Manager at Fujistu says,

> *"SMS phishing is dangerous to consumers because the end goal of attackers is nearly always to see enough data to complete fraudulent actions (most notably purchases in the consumer's name) or even blackmail if sensitive personal/employee data is obtained.*
>
> *The most concerning and common risk for businesses is the risk phishing proposes to breaking laws and legal agreements with suppliers/customers. For example, many business phishing attempts today target data that could be exposed as a leak of GDPR legislation; or for companies operating in say, financial or military industries, phishing can be used to obtain data that is by very strict regulation (or law) meant to not be seen out of employee's hands with the necessary security clearances. Such breaches can lead to serious legal, reputational, and financial damages." — Kaelum Ross*

- **SMS phishing damages your reputation and brand.** As **Reuben Yonatan** says,

> *"Customers expect organizations to protect them from fraud and data theft. They will not only blame the organization if a successful smishing attempt hurts them, but also seek to distance themselves from the company,*
>
> *A loss of trust from customers can easily crumble your business." — Reuben Yonatan*

[2] https://whatintech.com/

2.1.5. Examples

- **Generic or personalized messages attack**. The following compilation of smishing example screenshots come from Roni Bliss, Director of Sales of *The SSL Store*[3]. In some of the scam examples below, the attackers address her by her real name, while others are *"spray-and-pray"*[4] type phishing attempts. See figure 2.1.

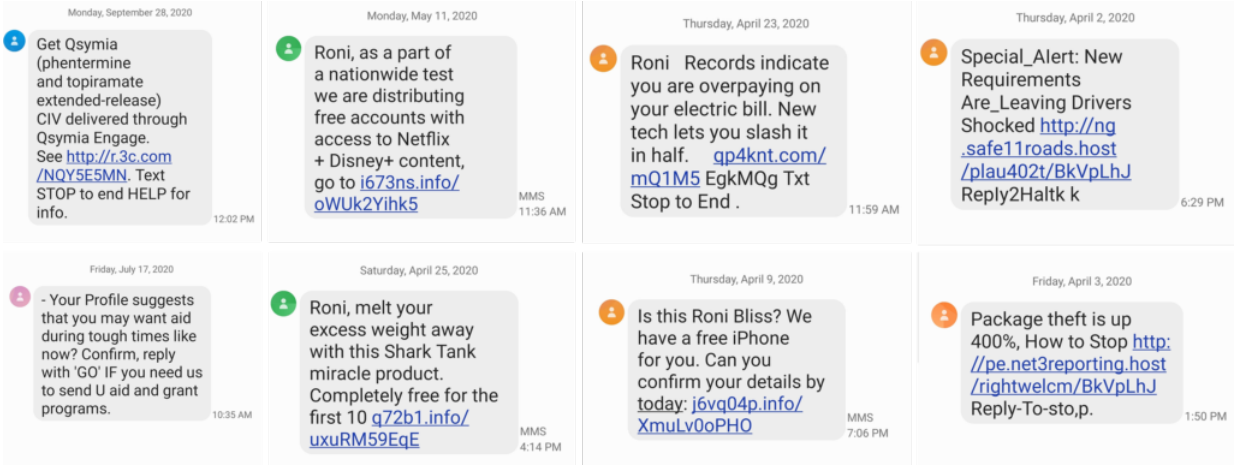

Figure 2.1.: Generic and personalized scam messages

- **Impersonate banks & other financial institutions attack**. Free money is something that rarely exists. SMS phishing scammers would love you to believe otherwise, so that you will click on their malicious links. This is an example of a smishing message purporting to impersonate US Bank. See figure 2.2.

[3] https://www.thesslstore.com/

[4] The applied idea of spray and pray is to send messages, out to as many people as possible (*spray*), and hope that it motivates some of them to take action or show interest in their requests (*pray*).

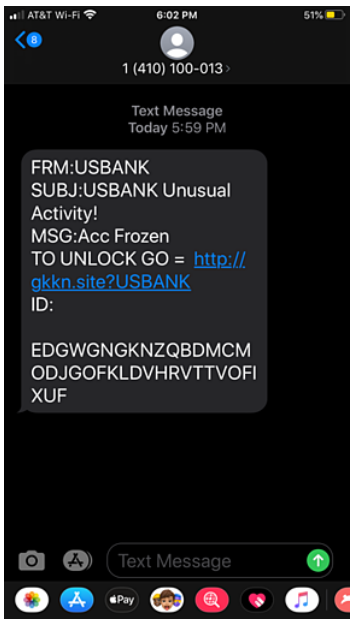

Figure 2.2.: Impersonate US bank scam message

- **The urgent message about your credit card attack.** In a similar way to the previous attack, in this case the attackers intended to alarm the recipient about the status of a credit card by impersonating the company American Express. See figure 2.3.

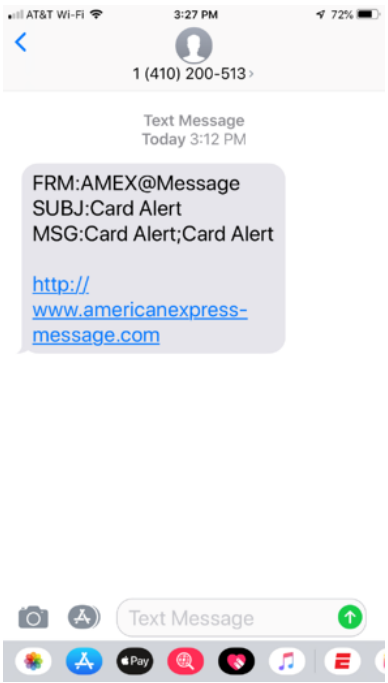

Figure 2.3.: Credit card American Express scam message

- **You won a prize and click here to get it attack.** These types of messages are tempting because you are receiving incredible rewards without doing anything, but there is no truth behind them. In this example they are pretending to be from Walmart. See figure 2.4.

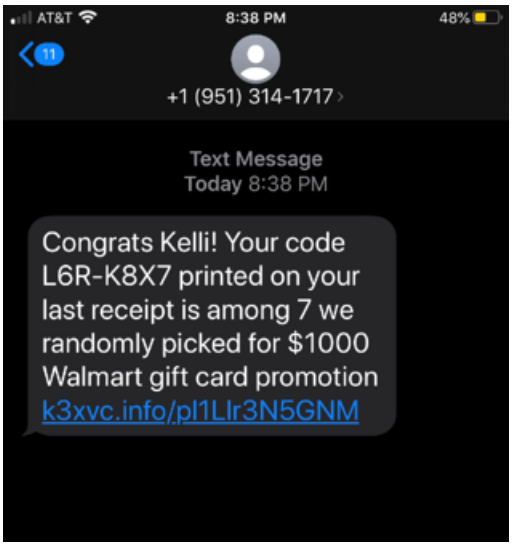

Figure 2.4.: Claim your rewards scam message

There are infinite examples of fraudulent messages, the limit is only in your imagination, but these are the most commonly used.

2.1.6. How to protect yourself and your organisation

How to protect yourself

- **Do not open text messages from unknown users.** If you receive a text message from an unknown number, particularly one that contains a link, **don't open it**.

- **If you open a text message, do not click on any links.** If you opened a text message from an unknown number and now this contains a link, **don't open it**. If you are sure that the message is legitimate, you can open your web browser and type there the website address.

- **If you click on a link, do not provide any information.** If you are asked to provide information and are redirected to a website, check for that website in your browser and verify that is legitimate by looking the website's security certificate information.

- **If you click on a link and provide information, take action.** Depending on the type of information you provide, this action may involve you to change your account security information (specially passwords), contacting with your bank or other financial services, cancelling your credit cards, etc.

- **Check if your phone has a filter to block messages from one source.** In order to avoid receiving messages from a suspicious source, some devices provides a fast way to block numbers.
- **Contact with your phone provider in order to block the malicious traffic.** Some cell companies provides tools to their customers to help you block smishing from known and unknown numbers.

How to protect your organisation

- **Provide cyber awareness training to all staff.** The first step in combating any type of cyber fraud is to educate your users about the different types of dangers that exist. This includes educating about phishing, smishing, vishing and other types of cyber threats.
- **Implementing a Bring Your Own Device (BYOD) policy.** Implementing this type of policy becomes meaningful when the company's employees use their personal devices for work. This policy outlines the rules to be followed when using personal devices and how the **I**nformation **T**echnology (IT) team will support them. It also helps mitigate security risks by controlling how your employees use those devices.
- **Bring only indispensable accesses.** Not everyone in an organisation needs access to all resources (databases, network and other critical systems). Limiting access to the sensitive resource just only for specific employees reduces the exposure to a potential attack should any of those involved be affected by a smishing or other attack.
- **Provide a way for customers to notify possible types of scams.** It is always a good idea to provide your customers with a space to give feedback on possible scams in order to cut off traffic from those sources.
- **Notify customers about potential smishing scams.** If you receive any information that someone is impersonating your organisation, you should warn your customers via email to be careful and follow the appropriate protocols to mitigate the possible consequences.

2.1.7. Conclusion

The types of text message scams that contain smishing are not new, they are not going anywhere as long as they continue to have an effect. Smishing is one of the areas that any organisation should cover and train its employees in order to raise awareness and avoid possible major consequences. This is exacerbated by the use of personal devices to perform work-related functions, exposing them to a greater risk of attack.

Cybercriminals are always looking for new ways to find potential victims and devise new types of attacks that can be effective. This is why it is necessary to act proactively and minimise the risk of threat not only to the organisation, but also to the customers who make use of the organisation's applications.

2.2. Prediction models

This section is intended to elaborate and obtain information about the different forecasting models used in the papers in subsection 1.2. For each type of model, its characteristics will be explained.

2.2.1. Multinomial Naive Bayes

What is

Multinomial Naive Bayes algorithm is a **probabilistic learning method** that is mostly used in **N**atural **L**anguage **P**rocessing (NLP). The algorithm is based on the Bayes theorem (See Figure 2.5) and predicts the tag of a text such as a piece of email or newspaper article. It calculates the probability of each tag for a given sample and then gives the tag with the highest probability as output.

Naive Bayes classifier is a collection of many algorithms where all the algorithms share one common principle, and that is each feature being classified is not related to any other feature. The presence or absence of a feature does not affect the presence or absence of the other feature.

How it works

Naive Bayes is a powerful algorithm that is used for text data analysis and with problems with multiple classes. To understand Naive Bayes theorem's working, it is important to understand the Bayes theorem concept first as it is based on the latter.

Bayes theorem shown in Figure 2.5, formulated by Thomas Bayes, calculates the probability of an event occurring based on the prior knowledge of conditions related to an event. Where we are calculating the probability of class A when predictor B is already provided. This formula helps in calculating the probability of the tags in the text.

$$P(A \mid B) = \frac{P(B \mid A) \cdot P(A)}{P(B)}$$

P(A | B)	the probability of A given B
P(B | A)	the probability of B given A
P(A)	the probability of A occuring
P(B)	the probability of B occuring

Figure 2.5.: Bayes theorem formula

Advantages

- **Easy to implement.** You only have to calculate probability.
- **Versatibility.** You can use this algorithm on both continuous and discrete data.
- **Simple.** It is simple and can be used for predicting real-time applications.
- **Scalable.** It is highly scalable and can easily handle large datasets.

Disavantages

- **Accuracy.** The prediction accuracy of this algorithm is lower than the other probability algorithms.
- **Not suitable for regression.** Naive Bayes algorithm is only used for textual data classification and cannot be used to predict numeric values.

2.2.2. Decision trees

What is

Decision **T**rees (DTs) are a **non-parametric supervised learning** method used for classification and regression. Decision trees learn from data to approximate a sine curve with a set of if-then-else decision rules. The deeper the tree, the more complex the decision rules and the fitter the model.

Decision tree builds classification or regression models in the form of a tree structure. It breaks down a data set into smaller and smaller subsets while at the same time an associated decision tree is incrementally developed. The final result is a tree with decision nodes and leaf nodes. A decision node has two or more branches. Leaf node represents a classification or decision. The topmost decision node in a tree which corresponds to the best predictor called root node. Decision trees can handle both categorical and numerical data. An example of a simple decision tree is show in Figure 2.6.

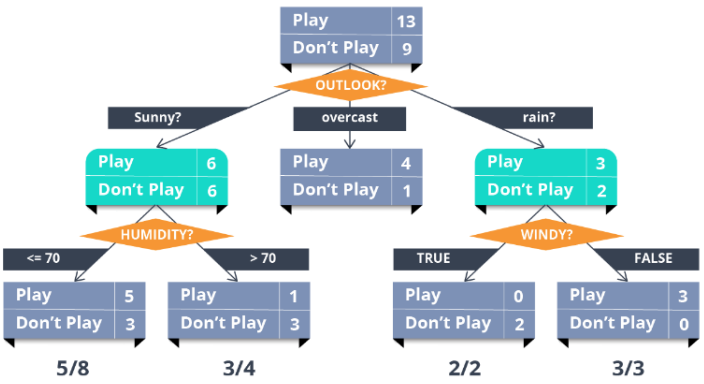

Figure 2.6.: Decision Tree example

How it works

There are several steps involved in the building of a decision tree:

1. **Splitting**: The process of partitioning the datset into subsets. Splits are formed on a particular variable. Figure **??** shows and example of splitting according two different variables: gender and class.

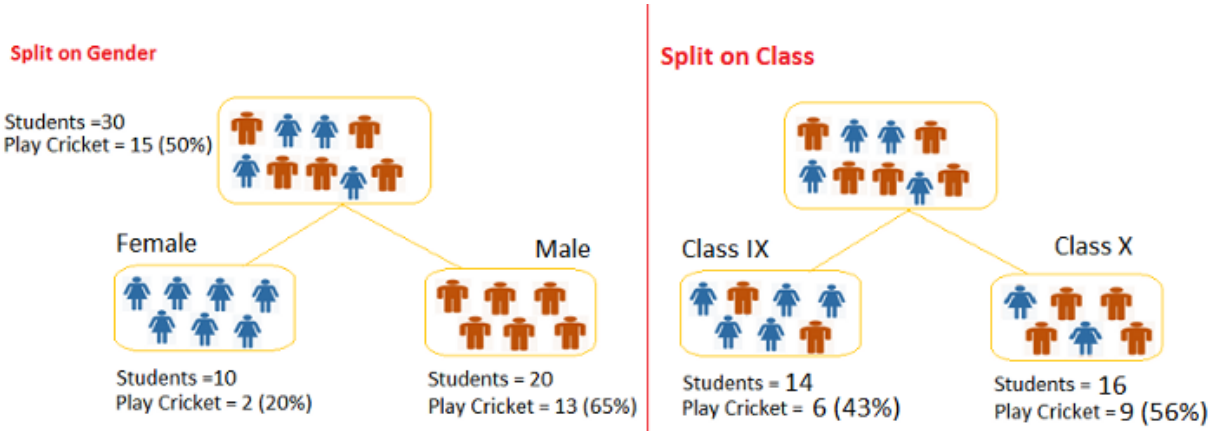

Figure 2.7.: Decision Tree splitting

2. **Pruning**: The shortening of branches of the tree. Pruning is the process of reducing the size of the tree by turning some branch nodes into leaf nodes, and removing the leaf nodes under the original branch. Pruning is useful because classification trees may fit the training data well, but may do a poor job of classifying new values. A simpler tree often avoids over-fitting. An example of that can be seen in Figure 2.8.

3. **Tree selection**: The process of finding the smallest tree that fits the data. Usually this is the tree that yields the lowest cross-validated error.

Advantages

- **Fast preparation.** Compared to other algorithms decision trees requires less effort for data preparation during pre-processing.
- **Data normalization not needed.** A decision tree does not require normalization of data.
- **Data scaling not needed.** A decision tree does not require scaling of data as well.
- **NaN values don't affect DTs build.** Missing values in the data also do NOT affect the process of building a decision tree to any considerable extent.
- **Understandable results.** A Decision tree model is very intuitive and easy to explain to technical teams as well as stakeholders.

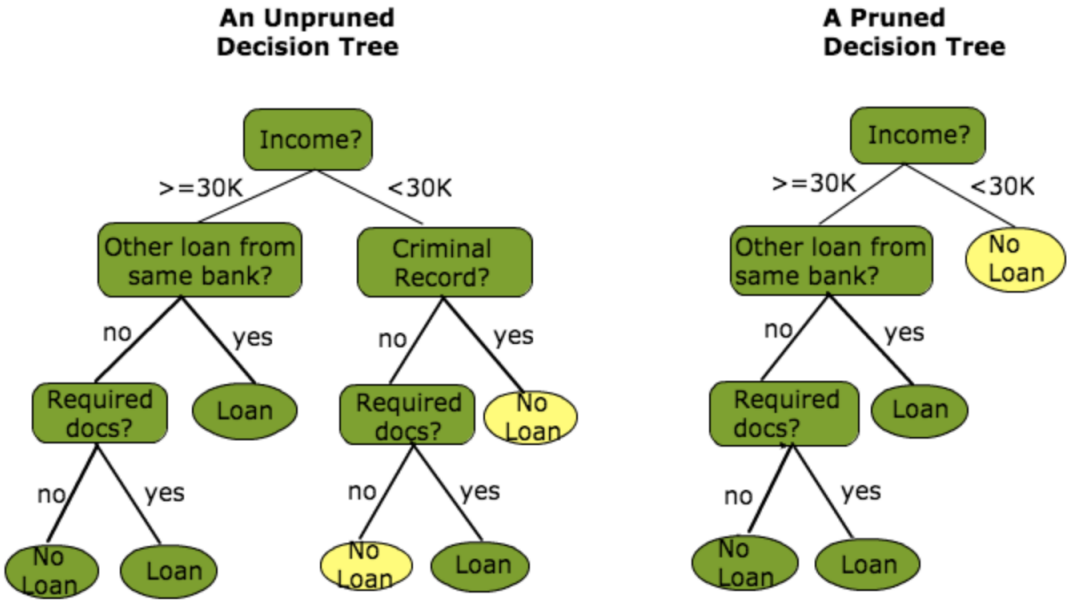

Figure 2.8.: Decision Tree pruning

Disavantages

- **"Butterfly effect".** A small change in the data can cause a large change in the structure of the decision tree causing instability.
- **Calculation complexity.** For a Decision tree sometimes calculation can go far more complex compared to other algorithms.
- **Higher training time.** Decision tree often involves higher time to train the model.
- **Expensive training process.** Decision tree training is relatively expensive as the complexity and time has taken are more.
- **Not suitable for regression.** The Decision Tree algorithm is inadequate for applying regression and predicting continuous values.

2.2.3. Ripper

What is

RIPPER by Cohen (1995) [6] is a variant of the Sequential Covering algorithm. RIPPER is a bit more sophisticated than Covering algorithm, and uses a post-processing phase (rule pruning) to optimize the decision list (or set). RIPPER can run in ordered or unordered mode and generate either a decision list or decision set.

RIPPER is based on classification rules, these represents knowledge in the form of logical if-else statements that assign a class to unlabeled examples. An "antecedent" and a "consequent" are the terms for them. This form a statement that says "if this happens, then that

happens".

The earlier rule learning algorithms (Separate and conquer, and The OneR[5] algorithm) have some problems like slow performance for an increasing number of datasets, and prone to being inaccurate on noisy data.

Johannes Furnkranz and Gerhard Widmer in 1994 proposed a solution towards solving these problems. The IREP algorithm uses a combination of pre-pruning and post-pruning methods that grow very complex rules and prune them before separating the instance from the complete dataset.

The RIPPER algorithm introduced by W. Cohen in 1995 improved upon IREP to generate rules that match or exceed the performance of decision trees.

How it works

Having evolved from several iterations of the rule learning algorithm, the RIPPER algorithm can be understood in a three-step process. Just like decision trees, the information gain criterion is used to identify the next splitting attribute.

1. **Grow.**
2. **Prune.**
3. **Optimize.**

The first step uses a *"separate and conquer"* method to add conditions to a rule until it perfectly classifies as a subset of data. Just like decision trees, the information gain criterion is used to identify the next splitting attribute. When increasing a rule's specificity no longer reduces entropy, the rule is immediately pruned. Until reaching stopping criterion step one and two are repeated at which point the whole set of rules is optimized using a variety of heuristics.

Advantages

- **Easy to understand.** As DTs, it's very intuitive and easy to explain the results to technical teams.
- **Representable in first order logic.** Can be implemented in languages like Prolog[6].
- **Prior knowledge.** Information about the problem available, in addition to the training data, can be added to the model easily.

[5] **OneR**, short for "One Rule", is a simple, yet accurate, classification algorithm that generates one rule for each predictor in the data, then selects the rule with the smallest total error as its "one rule".

[6] **Prolog** is a logic programming language. It has important role in artificial intelligence. Unlike many other programming languages, Prolog is intended primarily as a declarative programming language. In prolog, logic is expressed as relations (called as Facts and Rules). Core heart of prolog lies at the logic being applied. Formulation or Computation is carried out by running a query over these relations.

Disavantages

- **Poorly scalability.** Rule Sets scales poorly with training set size.
- **Noisy data.** Have problems with any data that cannot be understood and interpreted correctly by machines, such as unstructured text.

2.2.4. Prism

What is

Prism is a popular **rule-based algorithm**[7], which works with the concept of target class and is capable of selecting attributes based on their importance to a particular class [4].

The **Prism algorithm** was introduced by Cendrowska [3] in 1987. The aim is to induce modular classification rules directly from the training set. The algorithm assumes that all the attributes are categorical. When there are continuous attributes they can first be converted to categorical one. Alternatively the algorithm can be extended to deal with continuous attributes. Prism uses the **"take the first rule fires"** conflict resolution strategy when the resulting rules are applied to the unseen data, so it is important that as far as possible the most important rules are generated first.

How it works

The algorithm generates the rules concluding each of the possible classes in turn. Each rule is generated term by term with each term of the form *"attribute = value"*. The attribute/value pair added at each step is chosen to maximize the probability of the target "outcome class". The basic Prism algorithm is shown in Algorithm 1, described in [5].

Algorithm 1: Classical Prism algorithm

Input: A training dataset C with n classes, i = 1, 2, 3, ..., n
Output: Generated rules for all classes

1 **for** C_i *class in* C **do**
2 Compute the probability of each attribute/value pair with the complete training set for the class C_i
3 Select the pair with the largest probability and create a subset of the training set comprising all the instances with the selected attribute/value combination for each class C_i
4 Repeat steps 2 and 3 for this subset until a subset is reached that contain only instances of C_i
5 The rule is induced by the conjunction of all the attribute/value pairs selected
6 Remove all instances covered by this rule from the training set
7 Repeat step 2 through 6 until all instances of C_i have been removed

[7] *Rule-based learning* is an approach in which the model consists of a set of rules which were learned from the data [11].

2.2.5. Random Forest

What is

Random forest is a *supervised learning algorithm*. The "forest" it builds, is an ensemble of decision trees, usually trained with the "bagging" method. The general idea of the bagging method is that a combination of learning models increases the overall result.

How it works

One big advantage of random forest is that it can be used for both classification and regression problems, which form the majority of current machine learning systems. Let's look at random forest in classification, since classification is sometimes considered the building block of machine learning. In Figure 2.9 you can see how a random forest would look like with two trees. **Random forest** has nearly the same hyperparameters as a decision tree or a bagging

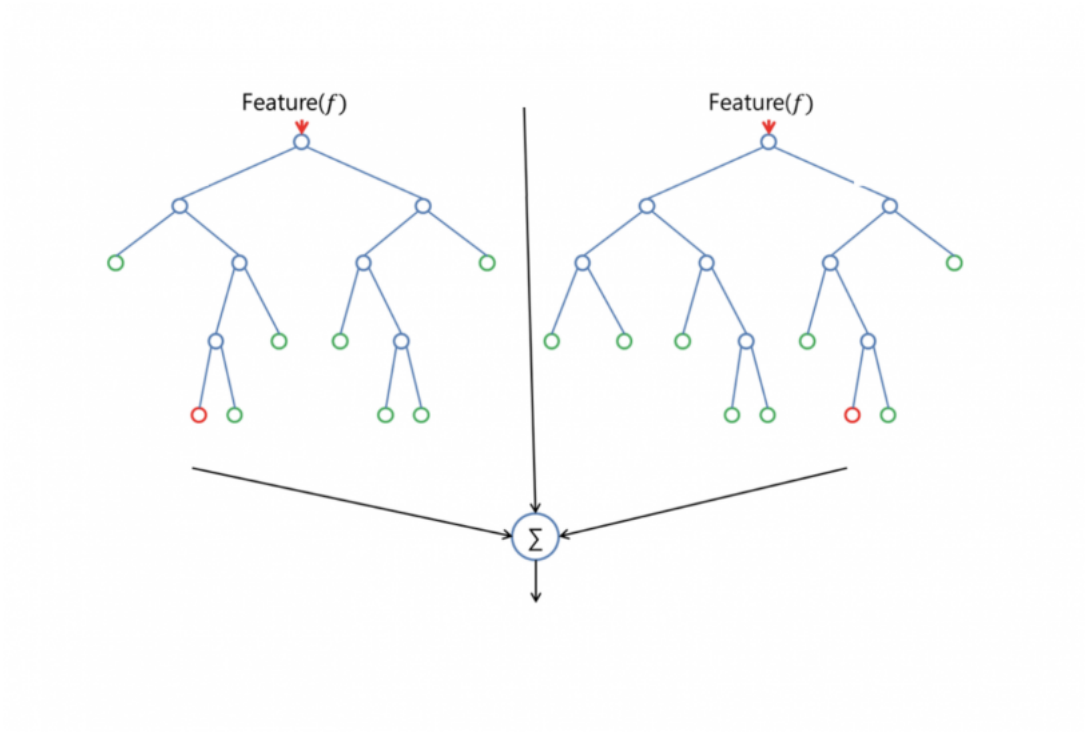

Figure 2.9.: Random forest with two trees

classifier. Fortunately, there's no need to combine a decision tree with a bagging classifier because you can easily use the classifier-class of random forest. With random forest, you can also deal with regression tasks by using the algorithm's regressor.

Random forest adds additional randomness to the model, while growing the trees. Instead of searching for the most important feature while splitting a node, it searches for the best feature among a random subset of features. This results in a wide diversity that generally results in a better model.

Therefore, in random forest, only a random subset of the features is taken into consideration by the algorithm for splitting a node. You can even make trees more random by additionally using random thresholds for each feature rather than searching for the best possible thresholds (like a normal decision tree does.

Advantages

- **Versatibility.** It can be used for both regression and classification tasks, and it's also easy to view the relative importance it assigns to the input features.

- **Very handy algorithm.** The default hyperparameters it uses often produce a good prediction result. Understanding the hyperparameters is pretty straightforward, and there's also not that many of them.

- **Avoids overfitting.** One of the biggest problems in machine learning is overfitting, but most of the time this won't happen thanks to the random forest classifier. If there are enough trees in the forest, the classifier won't overfit the model.

Disavantages

- **Efficiency.** A large number of trees can make the algorithm too slow and ineffective for real-time predictions. In general, these algorithms are fast to train, but quite slow to create predictions once they are trained. A more accurate prediction requires more trees, which results in a slower model. In most real-world applications, the random forest algorithm is fast enough but there can certainly be situations where run-time performance is important and other approaches would be preferred.

- **Predictive modeling tool.** Is a predictive modeling tool and not a descriptive tool, meaning if you're looking for a description of the relationships in your data, other approaches would be better.

2.2.6. Conclusions

To conclude the analysis and choose one base model to start with, we will analyze some features about them shown in Table 2.1. First of all, the accuracy. The accuracy of the [9] model is better than [7], staying close to the [13], but we have to take into account that all of those models have been trained using english datasets, we don't know at first how the behaviour would be whether we use other languages datasets so, we can't consider the accuracy as the single determinant metric.

Model	Smishing Detector[9]	Rule-Based Framework[7]	SmiDCA[13]
Classification approach	Naive Bayes	Decision Tree, RIPPER and PRISM	Random Forest, Decision Tree, Ada Boost and SVM
Accuracy	96.29%	95.02%	96.40%
Keywords classification	✓	✓	✓
Presence of URL	✓	✓	✓
Presence of phone number amd email id	✓	✓	✓
URL in blacklist	✓	✗	✗
Check for login page	✓	✗	✗
Difference in URL domain	✓	✗	✗
APK Download	✓	✗	✗
APK after redirection	✓	✗	✗
User consent while downloading APK	✓	✗	✗

Table 2.1.: Comparative between the different analyzed models

As we can see in Table 2.1, another important factor is that the *Smishing Detector* model provided by Mishra et al. [9] performs more security URL validations than the others, these grants an extra security to the system. Moreover the [9] model is very scalable, just adding more modules and verifications for each new feature we wanna consider.

The Multinomial Naive Bayes algorithm has so many applications in several industries, and the predictions made by this algorithm are real-quick. News classification is one of the most popular use cases of the Naive Bayes algorithm. It is highly used to classify news into different sections such as political, regional, global, and so on.

So, for our project we'll use the *SmishingDetector* model provided by Mishra et al. [9].

3. Requirements

Before the system is designed and implemented, it is essential to document what is expected to do. It is very important to discuss the needs and expectations of our end-user before we start development, to ensure that we are on the right track and to save rework. From a practical point of view, the goal is not to have perfect software requirements, but it is crucial to define the most important ones in order to start with the design and implementation.

The process of discovering, analysing, classifying and specifying requirements to be developed is called **R**equirements **E**ngineering (RE) [1]. This definition of requirements describes the various types of information that are captured by it.

3.1. Types of requirements

This definition is based on the fact that requirements can contain both the view of the end-user regarding the behavior as well as the internal properties that make the system suitable. In fact there are three levels of requirements that can be distinguished: ***business***, ***user*** *and* ***system requirements***. ***System requirements*** can be split up in ***functional*** *and* ***non-functional*** requirements [15].

Business requirements

These requirements include the benefits that the organization implementing the system wants to achieve. It describes the goals and added value of the system in regard to the organization, **Lleida.net** in this case.

User requirements

These requirements involve what the end-users should be able to achieve with the system and what activities the end-user is able to conduct using the system. This can for example be represented with user stories and use case diagrams.

Functional system requirements

These requirements specify what must be implemented so that the user requirements can be fulfilled. The plan for implementing the functional requirements can be specified in the system design.

[1] Requirements are a specification of what should be implemented. They are descriptions of how the system should behave, or of a system property attribute. They may be a constraint on the development process of the system [12].

Non-functional system requirements

These are all the requirements that do not fall into the category of functional system requirements. Often they are also termed supplemental or quality requirements as they specify operation attributes of the system rather than behavior. Examples of non-functional requirements are requirements regarding the accessibility, availability, compatibility, security and response time. The plan for implementing the non-functional requirements is specified in the system architecture.

To prioritize requirements, the **MoSCoW**[2] prioritization can be used. With this technique the requirements are categorized into four categories [14]. The idea behind the MoSCoW prioritization is that in agile development projects there is often no time to satisfy all requirements. Even though all requirements can be important, the most important features have to be implemented first to deliver the largest benefits of the system to the stakeholders. MoSCoW is an acronym for the four following categories:

Must have

These features are absolutely vital to have in the product. Satisfying these requirements is the minimum scope of every development project before launching the product.

Should have

These are still important features to include in the product but they are often not as time-critical as the must have features.

Could have

These are features that are nice to have, but not necessary for the software to function. When the time and resources are available, these can be implemented in the current development phase.

Won't have (but would like)

These are the features that are the least critical or perhaps not appropriate at this moment. These requirements could always be satisfied in a later development phase.

3.2. Process

The process of RE can be divided into various activities that each have their own methodologies and techniques [10]. Often these activities are incrementally repeated as more requirements are specified.

[2]The term MoSCoW itself is an acronym derived from the first letter of each of four prioritization categories: **M** - *Must have*, **S** - *Should have*, **C** - *Could have*, **W** - *Won't have*

Eliciting requirements

Often termed the first step of the requirements engineering process, eliciting requirements is about gathering initial information from stakeholders[3] in order to be able to formulate requirements. This information gathering can be done using techniques that are focused on individuals such as questionnaires, surveys and interviews, but also with more informal group elicitation techniques such as focus groups and workshops. Other techniques include analyzing existing documentation (usually from the organization), prototyping when there is a lot of uncertainty about the requirements and model-driven techniques to visualize missing information.

Modelling and analyzing requirements

After information has been gathered in the elicitation step, the requirements can be modelled and analyzed. This involves visualizing relations between requirements and classifying requirements into one of the earlier mentioned MoSCoW levels. The modelling techniques include enterprise, data and domain modelling.

Communicating requirements

After the requirements have been discovered and specified, the succeeding step is to communicate the requirements back to the stakeholders to ensure that the stakeholders and the developers all comprehend the requirements so far. The way that the requirements are documented is crucial as this needs to be understandable for all the stakeholders. The documentation technique is also important for later stages to trace back the requirements and be able to check if all requirements are met in the final software product.

Agreeing requirements

After all stakeholders understand the requirements, it is time to reach an agreement of the final requirements as requirements can sometimes conflict each other. This is done by prioritizing requirements in negotiations. It is preferable of course that unrealistic expectations and desires of stakeholders are already attenuated in earlier activities.

Evolving requirements

As the software design and development evolves, the requirements can change and new requirements can be added on top of the initial requirements as the requirements engineering process starts to make stakeholders think about what they want. This activity is to ensure that requirements are managed and if new requirements conflict with already existing ones, trade-offs are made regarding costs and benefits.

[3] An stakeholder is a person, organisation or company that has an interest in a given company or organisation. In this case the stakeholders will be the same company.

3.3. Approach

In this section we will detail the phases mentioned in section 3.2, so that we can get a better overview of the requirements of our system and proceed to the design and implementation phase of the system.

3.3.1. Eliciting requirements

Before we start to identify the requirements of the system, we need to find out who the stakeholders will be. This is the first step in gathering information.

Stakeholders are all persons or organisations that are impacted by the new product or in some way have an influence on the requirements of the product, in this case the real-time smishing detection system. In this exploratory research phase, the details of the design and implementation of the final product are not yet known. This phase is particularly focused on identifying the stakeholders, which will be the people or organisations that will make use of the product in one way or another or that have a strong interest in the project, as they are responsible for some kind of decision.

Initially the project was intended to have only one direct stakeholder, the company Lleida.net, more specifically a technical sub-department called Core, which is responsible for managing the traffic of incoming and outgoing SMS, among other things. Therefore, the requirements of the stakeholders are summarised in the needs demanded by this Core sub-department, which in turn have already been previously transmitted through the company's clients or other internal departments such as **I**nter**C**onnection e**X**change (ICX).

Once the stakeholders are known, we will use traditional techniques to determine the boundaries of the current system. For this work we will mainly rely on the following three:

1. **Introspection.** Introspection implies that we will first need to understand the main basic aspects of the system. This is often used as a starting point for other techniques.

2. **Reading existing documents.** This phase involves doing research on relevant aspects of our system by reading related documents.

3. **Meetings.** By using the meetings with stakeholders involved with the system we will be able to identify new areas for improvement of the current protocols.

Before modelling and analysing the system requirements in more depth, we need to understand the main aspects of the system, using this three traditional techniques for determining the system boundaries.

3.3.1.1. Introspection

We will start with an introspection to narrow down the main aspects that our system must fulfil. We will focus on the general ideas that this product must have, some of them are the following:

- The system **must be able to** detect, with a high degree of accuracy, text messages containing smishing.
- The system **must be able to** operate in real time.
- The system **must be able to** support a multi-language structure in order to be able to classify an SMS according to its language.
- The system **must be** optimised as much as possible to ensure the highest possible efficiency and performance.

3.3.1.2. Reading existing documents

Once the more general ideas of the project have been defined, we will focus on the reading of documents related to the topic, in this case related to smishing detection. All the analysis is described in the section 1.2 during the research on the current state of the art and in the chapter 2 where a more exhaustive analysis of the concepts of the project is carried out, from the very meaning of smishing to the description of the different types of predictive models currently used.

3.3.1.3. Meetings

Once the introspection and the reading of documents related to the project is done, we will proceed to the meeting with the main stakeholder Carlos Fernandez, Development Manager of Core's technical sub-department. The objective of this "informal" meeting is to extract new requirements and to specify what their needs or expectations are in order to start the design and implementation as soon as possible. The meeting will be held telematically and in an informal atmosphere. It should be noted that this meeting will also be attended by David Tapia, Development Manager of the technical sub-department of Apps Core, who will also provide a more experienced vision of the product and finish polishing the requirements of the system, as well as acting as project supervisor.

After this meeting, we have obtained new requirements and specifications, as well as a more concrete vision of which are the aspects that we must enhance and/or get the most out of it. We have been able to see that one of the biggest requirements we have been asked for is the validation speed, this is due to the fact that in order to integrate this validation in real time during the sending of a text message, we need it to consume as little time as possible so as not to overload the total time it takes to send a text message. This point has been critical, since no matter how efficient our system is, if we want to do a complete validation of the links containing messages, phone numbers and other elements to be validated, we depend on a network or database factor which in some cases could increase the response time of the service.

The solution to this problem involves the design and implementation of an asynchronous **RE**presentational **S**tate **T**ransfer (REST) service, so that validations on text messages can be launched in real time, but without having to wait for the results as a synchronous service. In this way, what we will achieve is not to avoid sending sms's, but we will be able to detect in the shortest possible time those text messages containing smishing and to block all traffic

coming from those sources.

Another aspect that was discussed during this meeting was the saving of certain data about each message in the database. The Core guys need to send and receive certain kind of information to be able to identify each message that we have already validated, for each request we will have to save the identifier associated to each message and return it once we have been asked for the validation response.

To avoid constant polling[4] to obtain the results of the asynchronous validation, it was proposed to implement a notification functionality via callbacks, so that when the validation of a text message is finished, it will be notified through the callback url of the completion and its result.

Last but not least, we have emphasised the issue of efficiency in detecting messages that may contain smishing. It is very important for our system to be as accurate as possible, to prevent Lleida.net customers from receiving these messages.

3.3.2. Modelling and analyzing requirements

At this point, where all the information from the eliciting requirements step has been collected, we can model and analyse the system requirements. As previously mentioned, this step consists of visualising all the relationships between the requirements and classifying them according to their types:

- **B**ussiness **R**equirements (BR)
- **U**ser **R**equirements (UR)
- **F**unctional **R**equirements (FR)
- **N**on-**F**unctional **R**equirements (NFR)

It also assigns them a MoSCoW priority type (See 3.1 for more info) according to the type of need. During the whole process many options and new priorities have been discussed, therefore these have also been added to the list of requirements even though they are not part of the first line of implementation, but can be implemented in the future. These requirements are shown in Table 3.1.

[4]Polling in computing refers to a constant polling operation, usually to a hardware device, to create synchronous activity without the use of interrupts, although this can also be the case for software resources. In this case, it would be the constant demand for validation results.

#	Requirement type	Requirement description	MoSCoW priority
1	*BR*	Detect smishing messages with a high degree of accuracy in order to stop traffic from those senders.	Must
2		Supporting the system in case of possible failures or upgrades.	
3		Offer an extra layer of security to the SMS service.	
4		Minimise system response time.	
5		Validation of English and Spanish text messages.	Should
6		Make a product that is portable and can be used individually as a validation service.	Could
7		Add support for different languages in addition to English and Spanish.	Won't
8	*UR*	Validate text messages against smishing via a REST API.	Must
9		Get message validations asynchronously.	
10		Return message identifiers for subsequent identification of messages.	
11		To optimise service response time as much as possible.	
12		Return the response in a clear and concise format.	Should
13		Notification of completion of validations via Callbacks.	
14		Write clear documentation to implement REST API calls.	Could
15	*FR*	Implement a web service that works as a REST API.	Must
16		Develop a system to receive requests, validate them and return the results asynchronously.	
17		Add and store all the information necessary to identify a message.	
18		Implement a concurrent request validation system to process as many requests as possible at the same time.	
19		Implement a system that can work with multiple language types and is scalable.	
20		Use profiling tools to optimise the code and/or possible extensions to increase performance.	
21		Produce a JSON response with all data clearly specified (identifiers, code and alias of the request status, validation result, ...).	Should
22		Implementing a system for sending notifications via HTTP Callbacks.	
23		Write clear documentation to implement REST API calls and with indispensable system information.	Could
24	*NFR*	Analysis of the language to be used for the implementation of the system.	Must
25		Ensuring that all OWASP security recommendations are achieved.	
26		Perform unit and functional tests for each class or module of the system.	
27		Setting up virtual environments for development.	Should
28		Comment the code properly in order to be able to generate documentation of the code.	
29		The system has to be easy to integrate into a remote server.	
30		Use a version control system for the project (Git, subversion, cvs, ...).	

Table 3.1.: System requirements classified by types and MoSCoW priority

3.3.3. Communicating and agreeing requirements

Now that we have the requirements defined and classified by type and priority, it is time to transfer them to the stakeholders to confirm that the needs that they had previously communicated to us and those that we have interpreted are the same. Given that the needs requested, from the most general to the most specific, it has been confirmed that all of those requested requirements in the meeting detailed in section 3.3.1.3 are correct. On our side, we have also added some requirements that we had not initially thought of, so that the list is as complete as possible and we can start with the design and implementation of the system.

3.3.4. Evolving requirements

To date, no new requirements have been received for the system. For this step it would help to have a first version of the system already running so that the stakeholders could test the functionalities and new ones could be born. Therefore, we will leave this section as a future work to be done.

4. Design, Implementation and Evaluation

This chapter describes the design, implementation and evaluation of the system. Since it is an iterative process, we will make a first version that fulfils the requirements of the previous chapter. This version will have the objective of clearly detailing the structure of the system design, as well as its most important parts for its correct understanding. Once the design and implementation is done, the system will be evaluated in terms of speed, resources, etc. This evaluation has the purpose of extracting a series of improvement points that will be improved using some optimization tools.

4.1. Design

In this subsection, we will focus on explaining and detailing the structure of the system, as well as the connections between classes and modules, providing the design decisions taken.

As it's shown in Figure 4.1, the system will mainly consist of 5 main modules or blocks:

- API
- Core
- Database
- Local storage
- Modules

Each module has its function within the system, the combination of all modules is crucial for the system to work properly.

For our system we'll have mainly two flows, the first one to post validation and the second one to retrieve the results:

1. **Post request validation**: Send a request for the corresponding validation. See Figure 4.2.
2. **Get validation results**: Retrieve the validation results for an specific request. See Figure 4.3.

In order to control the current status of the requests an state machine is defined as in Figure 4.4.

A more detailed explanation of how each module works is given in the below subsections.

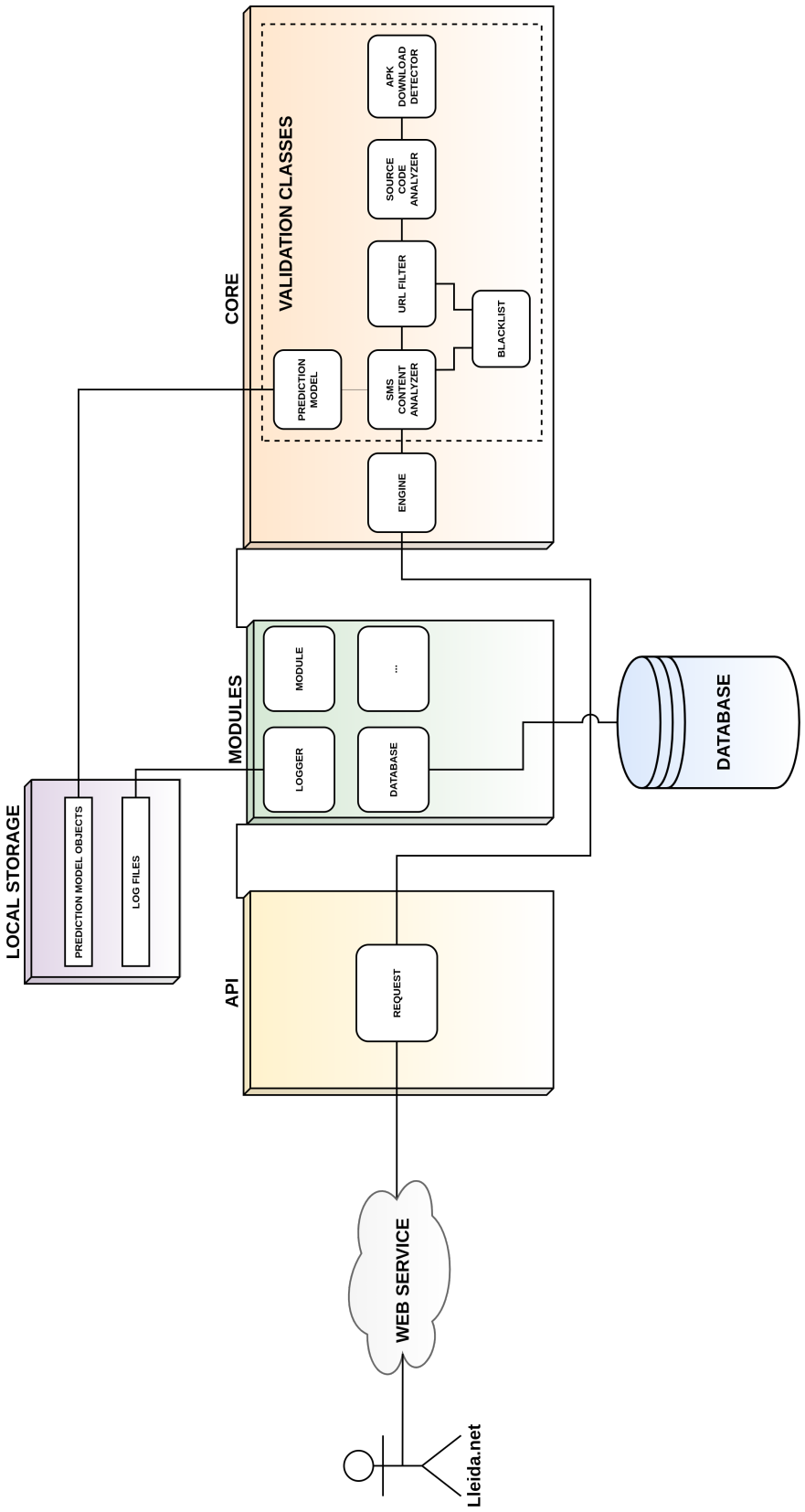

Figure 4.1.: System design architecture

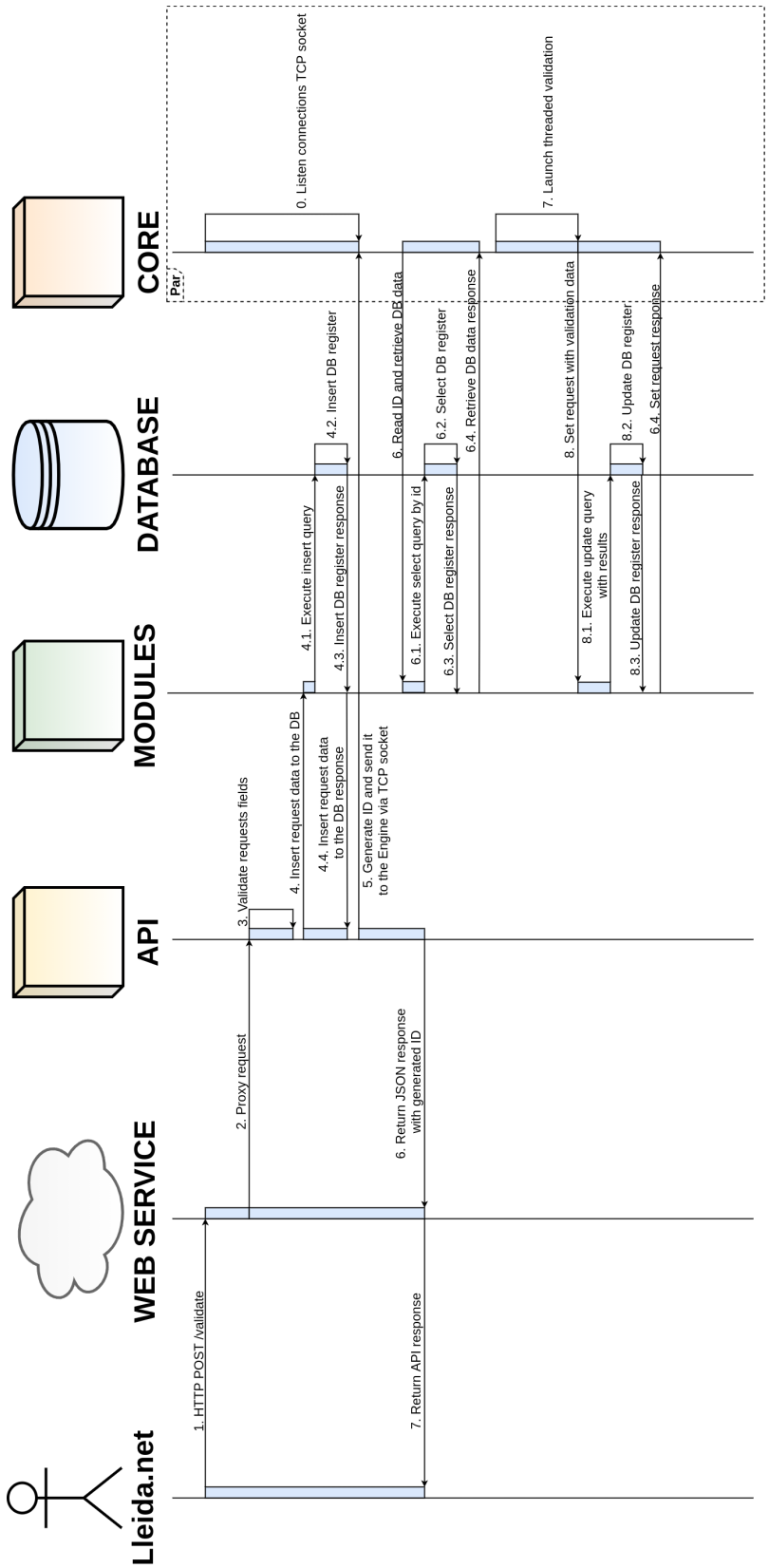

Figure 4.2.: POST validate sequence diagram

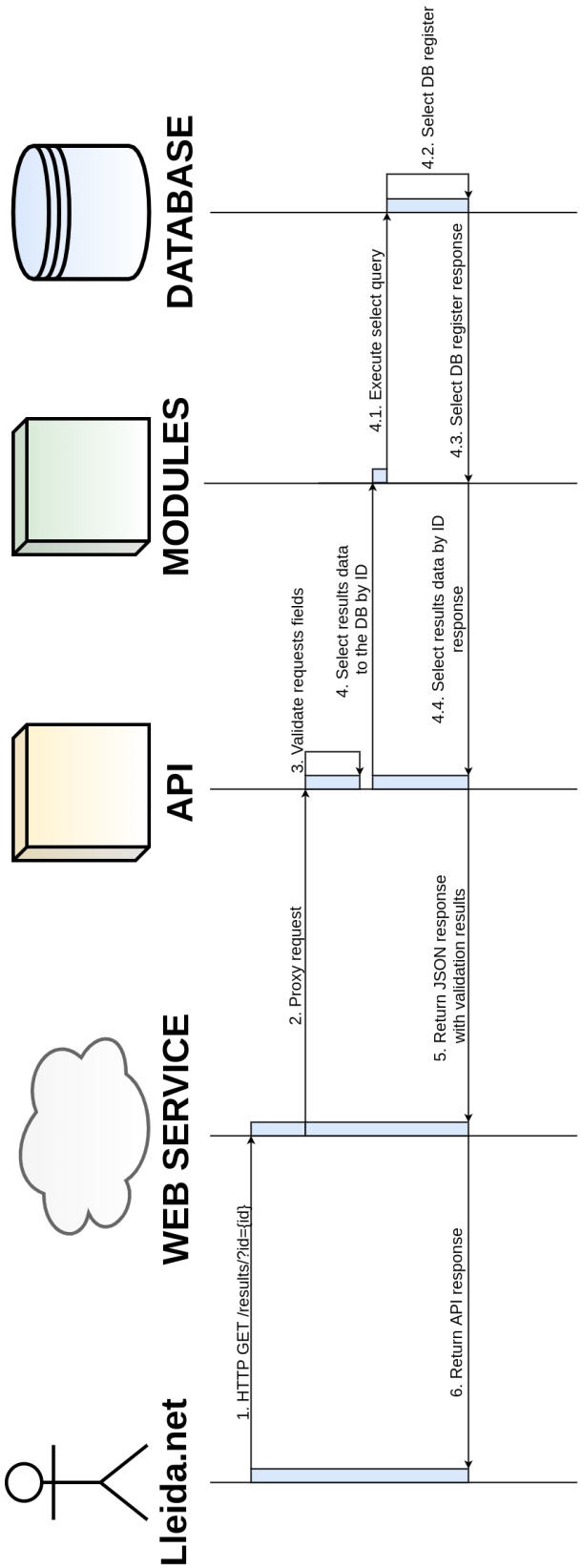

Figure 4.3.: GET results sequence diagram

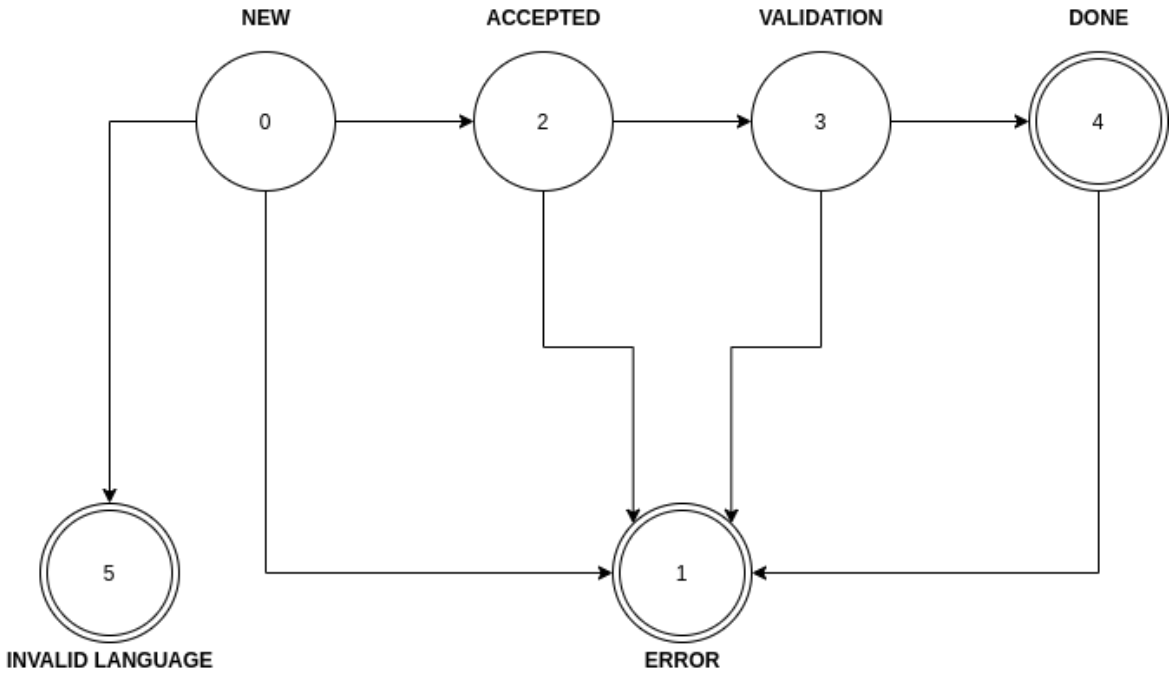

Figure 4.4.: System state machine

4.1.1. API

The API module will contain all the logic behind the communication with the web service, mainly the backend of the application that will receive, process and validate all API requests. For each request received, validations will be performed on the **J**ava**S**cript **O**bject **N**otation (JSON) fields of the request as well as the rest of the parameters. Finally, a unique request identifier will be generated and all the validated data will be inserted into the database. Once the record has been inserted in the corresponding table of the database, the previously generated identifier will be sent to the Core, so that it can retrieve the data from the database and perform the validation.

4.1.2. Core

The Core module will be in charge of managing all the message validation logic. For the moment we will differentiate between the following classes and functionalities:

- **Engine.** Acting as the brain of operations, this class will be in charge of receiving the identifiers sent by the API module and retrieving all the data associated with them from the database. Given that one of the main requirements of the system is that it is capable of validating messages in multiple languages, it must distribute the tasks to the subsequent **validation classes** according to the language detected. That is why for each different language we will have different validation objects. The Engine will also manage the change of states of the requests in the database and in a possible future it will also manage the sending of Callbacks.

- **Validation classes.** This group consists of the 4 classes or layers of the validation system and two auxiliary classes to make life easier. Therefore, the first group of classes for the validation of the system is composed of the following:
  1. **SMS Content Analyzer.** It will be the **first layer**, in charge of validating the content of the message. This class will contain within it the prediction model that will be used to predict whether the message is legitimate or not.
  2. **URL Filter.** It will be the **second layer**, basically in charge of validating a series of aspects about the url's contained in the message and will determine whether the message is legitimate or not.
  3. **Source Code Analyzer.** It will be the **third layer**, in case the second layer does not detect anything strange, for each effective url we will proceed to validate the source code of these to determine if the message is legitimate or not.
  4. **APK Download Detector.** Finally the **fourth layer**, if in all the previous ones we have not detected anything strange, we will proceed to validate if within those url's there is a download link with a potential apk application, which points to be a possible malware. If it does not contain it, we will continue validating the content of the other urls, otherwise we will mark the message as illegitimate.

  Finally we are left with **two** classes that exist to complement certain functionalities of the validation classes, these are:

  – **Prediction Model.** This class will be in charge of implementing all the logic related to the prediction models, this implies reading the models depending on the language, converting the message text into an input format for the model and returning the prediction results with the corresponding format.

    To avoid adding training and feeding time we will use a previously trained model and load it into memory as an object.

    This class will be only instantiated by the class **SMS Content Analyzer**, which will allow working indirectly with the prediction model through this same object without worrying about the internal logic.

  – **Blacklist.** The Blacklist class will also serve as an abstraction bridge in the manipulation of blacklists against the database. Therefore, it will contain different methods to be able to query, update and delete records from the database by means of methods of the class.

    This class is designed so that the blacklists will be dynamic, that is to say, as we detect malicious traffic we will be able to feed our blacklists with this information. This functionality is not critical and therefore, in a first version a static blacklist will be implemented, although the design will be made to be able to migrate it to a dynamic blacklist.

    The classes that instantiate this object will be **SMS Content Analyzer**, to work with phone and email blacklists, and **URL Filter** to work with url blacklists.

For the moment we have described in a general way how the Core is going to work, in the implementation phase we will see in more detail how to develop each of the described functionalities.

4.1.3. Database

The Database module will be of vital importance in our system, as it will be in charge of the data storage of all requests as well as all the information regarding the blacklists. Based on the requirements and design of the system, the database schema defined will be composed of 4 main tables:

- **request.** It shall store all the data concerning the request such as: the unique identifier, MT identifier (Lleida.net SMS identificator), the status, date received, date completed, string message text, message language, etc.
- **phone_blacklist.** It will act as a container for all **phones** that we consider to be fraudulent.
- **email_blacklist.** It will act as a container for all **emails** that we consider to be fraudulent.
- **url_blacklist.** It will act as a container for all the **URLs** that we consider fraudulent.

4.1.4. Local Storage

One of the most important aspects when developing and maintaining a product is the saving of logs, all that information regarding the behaviour of our system that allows us to identify problems in the code or operations as well as new improvements to increase the quality. That is why we will set up a system for saving logs in local files, being able to differentiate for different types of modules in which file they are written.

For prediction models, it would be very inefficient to create the whole model from scratch every time a restart is done, so the idea of saving the objects associated with the trained models makes sense. The loading time of these objects will be negligible compared to having to retrain them from scratch.

Many times local storage allows us to gain that extra speed that our system needs, especially with the saving of files and response, that is why it is indispensable for our system to have it.

4.1.5. Modules

This module contains all the generic classes that will be used in different parts of the system. All these generic classes will be inside the same module, which we will be able to import from any other class in order to instantiate them as objects and obtain their functionalities. At the moment we have planned to have 3 different classes, it is not to say that more will be added as new requirements come out. These classes are:

- **Database.** It will be in charge of establishing all the corresponding logic to create connections and perform operations against the database. The class that wants to use these functionalities will only have to instantiate an object of this class and call its methods.
- **Logger.** It will be in charge of providing the functionality of writing inside a specific file with a specific log format and a priority (**DEBUG**, **INFO**, **WARNING**, **ERROR** or **CRITICAL**) to the class that instantiates it. This class will interact directly with the Local Storage.
- **module.** This class is intended to provide all generic methods, libraries and constants to the importer. For example, it will contain database configurations, local storage paths, get a unique identifier, etc.

4.2. Implementation

Now that we are clear on how the system should be structured, it is time to decide on more specific aspects of the implementation, from the programming language to be used to the type of database.

4.2.1. Project setup

4.2.1.1. Language

One of the fundamental aspects in the implementation is to choose the programming language that we will use for our system. In this particular case, the language we will use to develop and complete all the requirements set out in previous chapters. To make this decision, we will have to consider several aspects such as the speed of the language, its functionalities and the degree of experience we have with it.

For this particular case I have chosen **Python**[1], more specifically **version 3.8.6**. It is always said that Python is not the fastest language when compared to other languages such as Java, C, C++, Go, etc. This is a fact, Python is a mostly slow language due to its dynamic nature and versatility. But what Python really offers us is a wide range of tools for all kinds of problems, where they are highly optimized and faster than other options.

Another main reason for using Python is that we have many tools that isolate the full complexity of artificial intelligence, and since in this project we will make use of them to integrate them directly into the system, we are very interested in simplifying the implementation as much as possible.

As previously mentioned, another important aspect to take into account is the degree of experience, Python is currently the language with which I feel most comfortable programming and with which I have the most experience, so I am very familiar with its nooks and crannies. One of those nooks and crannies that will make the difference in the efficiency of

[1] **Python**: https://www.python.org/

the code will be the optimization libraries, especially we will focus on one called **CPython**[2]. This library will allow us to statically define the variables of our python code in a simple way and increasing the efficiency in the critical regions of the code enormously.

4.2.1.2. Virtual environment

Usually when we work on projects where we have to install different types of libraries and these in turn may have dependencies with the operating system libraries, it is interesting to use an isolated development environment where we can install all these libraries without having to worry about breaking anything. Whenever possible, we should work with virtual environments or containers, which are also easily replicable. In this case we will use the tool **virtualenv**[3], it is a Python virtual environment that, as previously explained, will allow us to work in an isolated environment to be able to install libraries and runtime environments.

4.2.1.3. Version control software

To try to keep track of the elements and parts that make up the system, we will use a version control system. There are different types on the market, such as Git, **S**ub**v**versio**n** (SVN), Mercurial, etc. In this project we will opt for **Git**[4], since I currently have private repositories and it is the one I normally use to start all my projects.

4.2.1.4. Source-code editor

There are many tools on the market to develop in a comfortable and efficient way, such as **V**isual **S**tudio **Code** (VSCode)[5], PyCharm, Spyder, Sublime Text, etc. For now, the one that has given me the best results has been VSCode, since it offers many extensions to work with all the elements that our project has, from the programming language to the version control.

4.2.2. API

The flowchart 4.5 details the two types of petitions or flows of the Request class. The methods of the API have been implemented using the framework **Flask**[6], which allows us to quickly and successfully create the API code and then easily deploy it. At the moment we will have the two methods in the APIs explained above in more detail.

4.2.2.1. HTTP POST /validate

The left side of Figure 4.5 correspond to the initial flow, the one that will have to be called when you want to create a new SMS validation. So, as we can see in Figure 4.6 this API method works with **application/json** **M**ultipurpose **I**nternet **M**ail **E**xtensions (MIME) types and the request body is conformed by the following three attributes:

[2]**CPython**: https://cython.org/
[3]**virtualenv**: https://virtualenv.pypa.io/en/latest/
[4]**Git**: https://git-scm.com/
[5]**Visual Studio Code**: https://code.visualstudio.com/
[6]**Flask** :https://flask.palletsprojects.com/en/1.1.x/

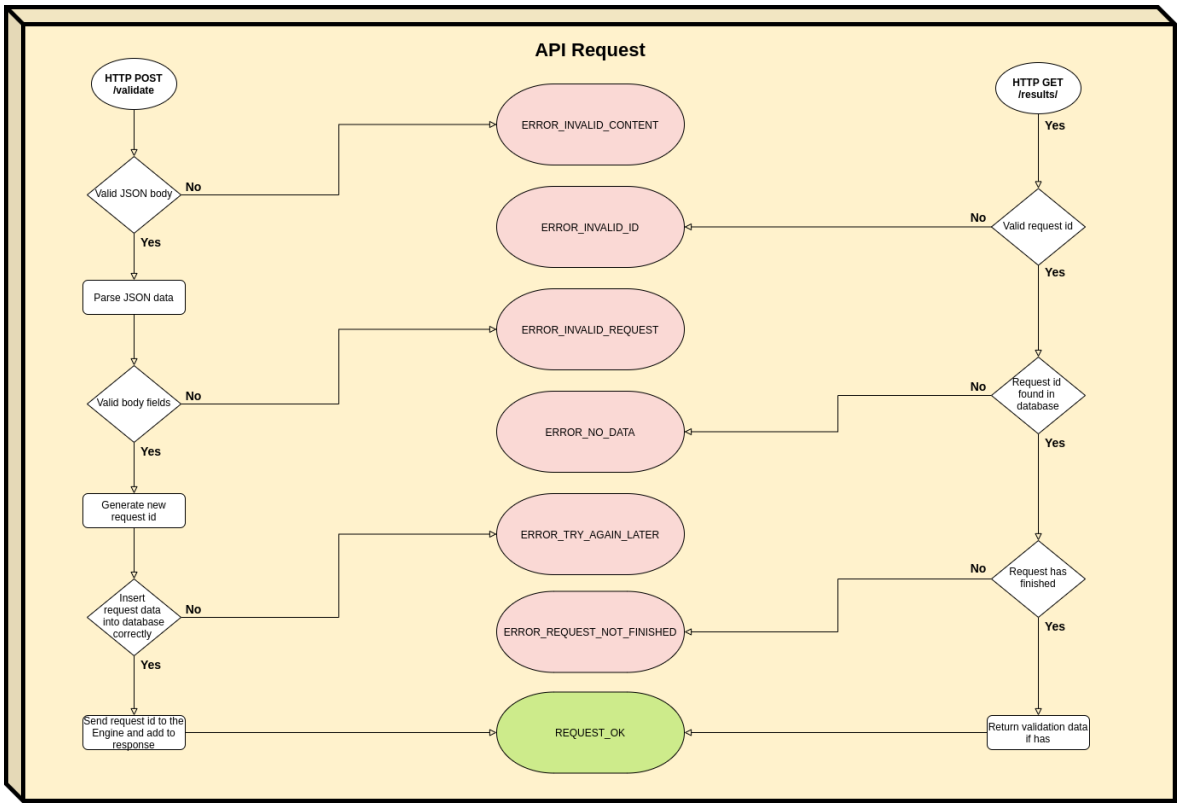

Figure 4.5.: API Request flowchart version

- **id_mt**: SMS Lleida.net identificator.
- **id_user**: Identifies an specific user.
- **text**: SMS text to validate.

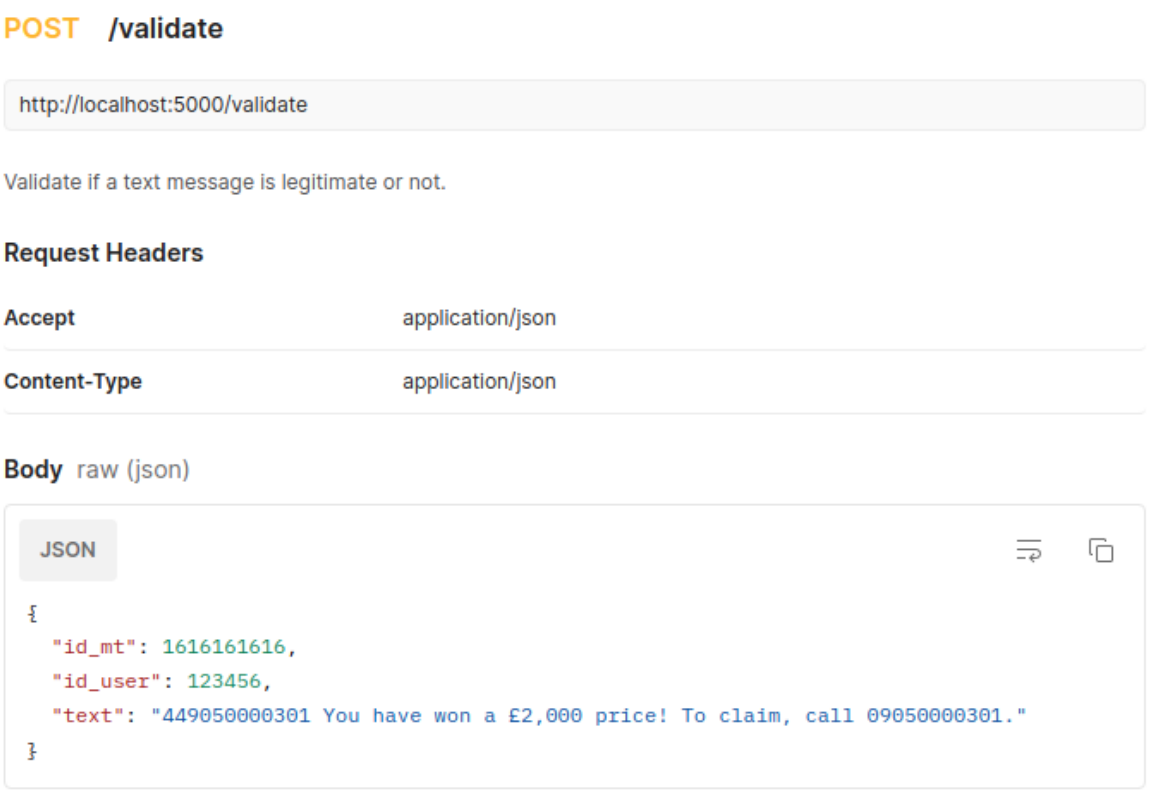

Figure 4.6.: API validate request example

Within this flow, validations are first performed on the JSON body, followed by the validation of the JSON attributes, generating the unique identifier of each request, inserting all the necessary fields into the database and sending the unique identifier through a TCP socket to the Engine. Each operation is dependent, i.e. if there is a problem, the corresponding error is returned and the request does not continue to progress. Finally, as we can see in Figure 4.7 a JSON is returned containing the following attributes:

- **code**: HTTP status code.
- **status**: HTTP status name.
- **id**: Request validation id to retrieve the results later.

4.2.2.2. HTTP GET /results/?id=

The right side of the Figure 4.5 correspond to the second flow, the one to use once launched the **/validate** API method and obtained a successful response including the SMS request

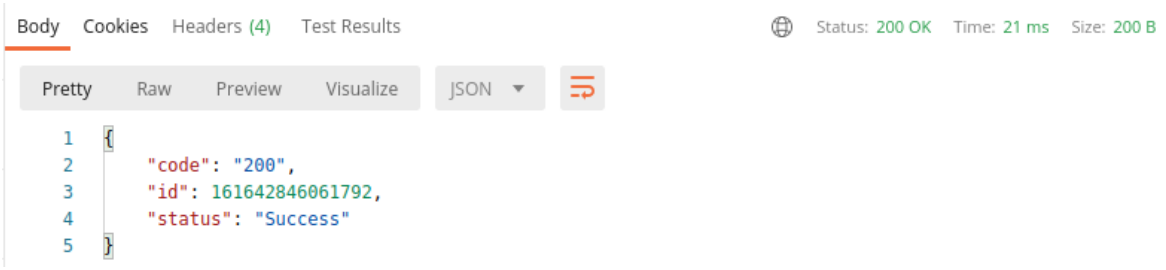

Figure 4.7.: API validate response example

identifier.

In order to retrieve the validation results of a previous request, we'll pass a GET parameter the id, returned by the first API method **POST /validate**, as shown in Figure 4.6.

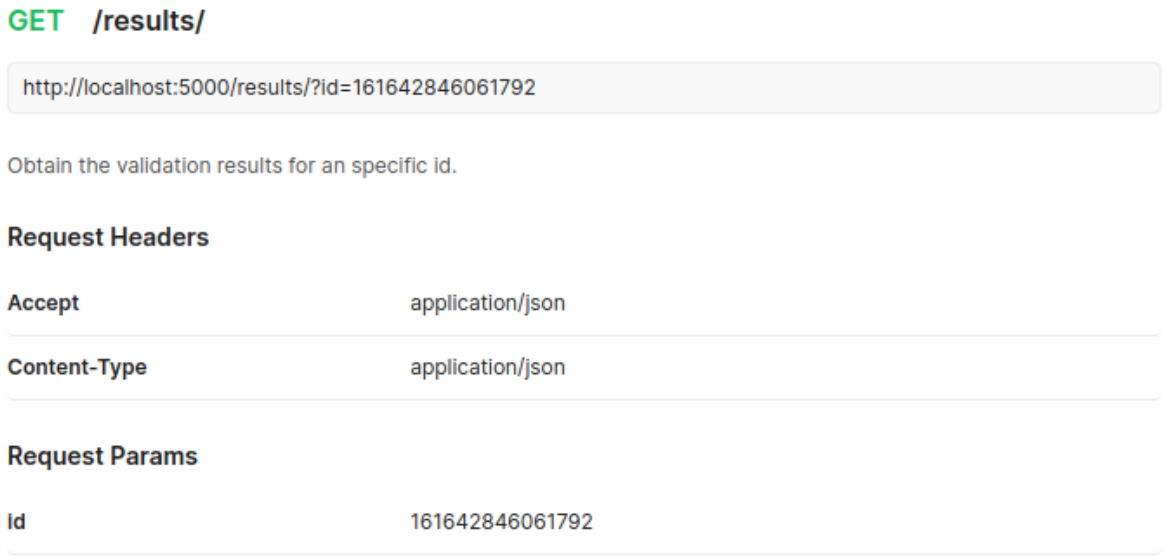

Figure 4.8.: API results request example

Then, the API validates that the request identifier is correct and try to retrieve the information associated with that identifier from our database. In case the request has not finished, the corresponding message is returned clarifying that the request has not finished yet. Finally, if the request has finished, all the information corresponding to the validation of that message is returned. The finished response shown in Figure 4.7 will contain the following attributes:

- **code**: HTTP status code.
- **status**: HTTP status name.
- **id**: Request validation id.

- **id_mt**: SMS Lleida.net identificator.
- **request_status_code**: One of the statuses in the Figure 4.4.
- **legitimate**: Only returned if the request has finished (request_status_code is 4). The value will be 0 if SMS is **ILLEGITIMATE** or 1 if is **LEGITIMATE**.

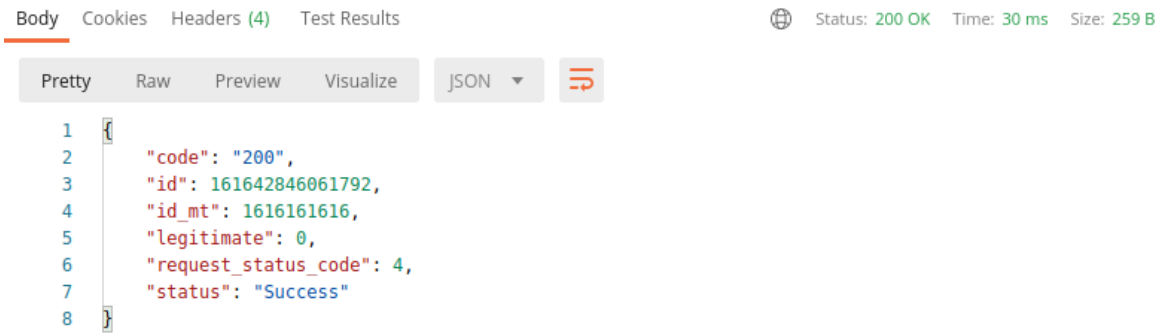

Figure 4.9.: API results response example

4.2.3. Core

As explained in the design phase 4.1, the Core of our system is composed of different classes or modules, all of them accomplishing some specific function. Below we explain how each of the different Core classes or modules have been implemented.

4.2.3.1. Engine

We will start by describing how the Engine works, which is responsible for receiving the identifiers of the requests to be validated and assigning them to the corresponding validation module, depending on the language of the message. In figure 4.10 we can get a visual explanation about the flow.

Mainly we can divide the work of the Engine in 3 main phases:

1. **Start**: In the **first phase**, we initialise all the class variables that will be needed during the execution of the class and then call the function to listen for requests (**second phase**).
2. **Wait**: In the **second phase**, we'll open the TCP socket connection to start listening to the API Request requests, and thus receive the request identifiers with which we can retrieve the data from the database necessary for validation. For each message to be validated we check that its language is accepted by the Core, otherwise the validation would not be coherent and the results would be affected. If the language is NOT accepted, we update the status of the database identifier of the request with

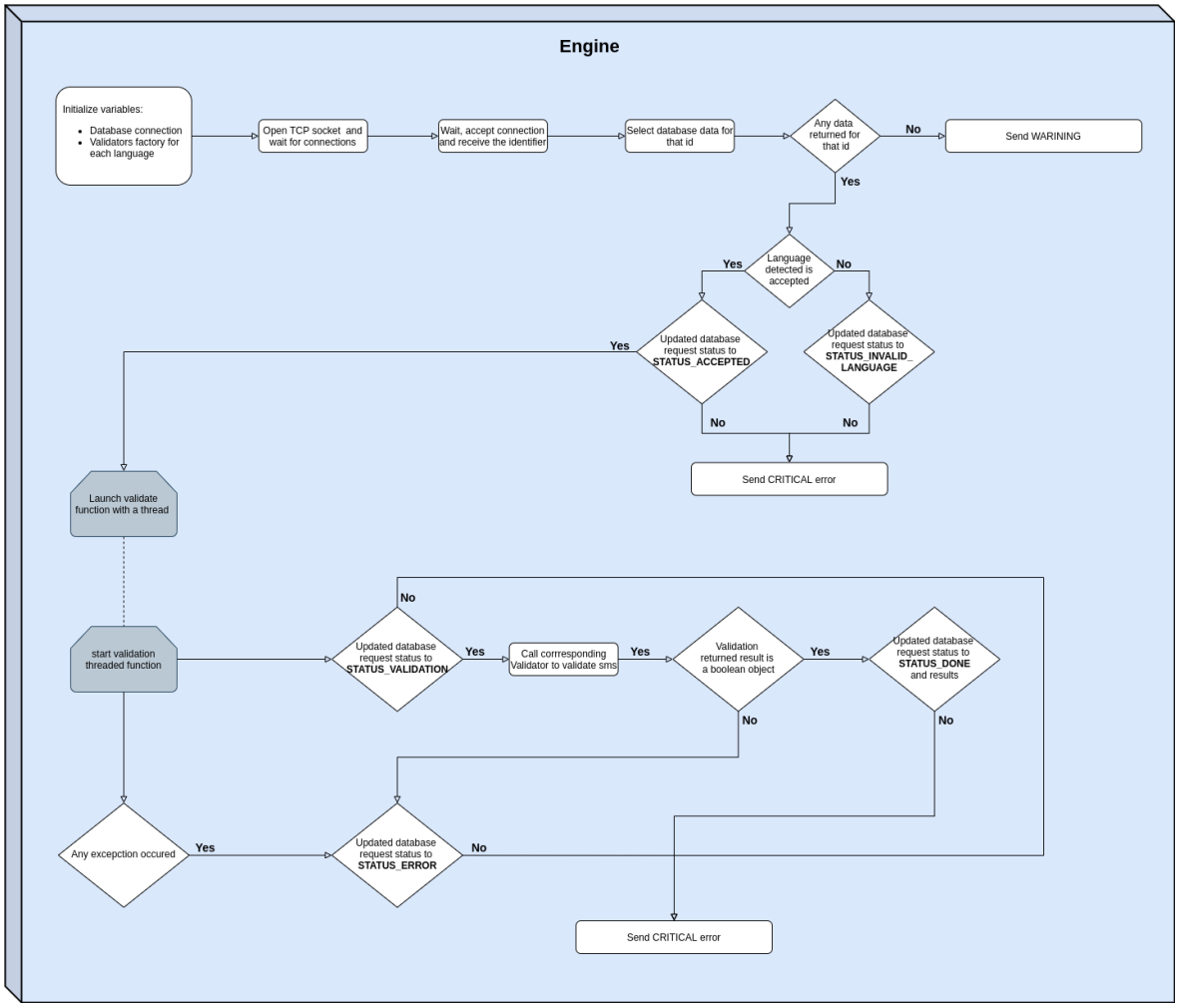

Figure 4.10.: Engine flowchart

STATUS_INVALID_LANGUAGE. Otherwise we will update the status to **STATUS_ACCEPTED** and we will launch a thread to validate the message, in order to continue listening and validating requests (**third phase**).

3. **Validate**: This phase is the **third phase** and is carried out concurrently to the previous phase. We will start by updating the status of the request to **STATUS_VALIDATION** and we will call the corresponding validator module according to the language of the message. This module will return a boolean, which will be **True** if the message is legitimate (does not contain smishing) or **False** if the message is illegitimate (contains smishing). Once we have the result of the validation, we update the status of the request to **STATUS_DONE** and the column of the database that contains the result of the validation, here the flow of the thread in charge of that validation will end.

   It is important to note that whenever there is an error or exception, we will update the status of the request to **STATUS_ERROR**.

4.2.3.2. SMSContentAnalyzer

This class is the **first layer** that will perform a validation on the text content of the SMS. The flowchart is represented in Figure 4.11.

The validation starts by applying a regular expression to detect URL or SAL in the content of the SMS. If there are any matches we pass them to the URL Filter and return the result of this, otherwise we continue with the validation. The next step is to check if the message contains email or phone numbers, here we will also follow the same strategy and apply two regular expressions, one for each case. If we don't detect any email or phone numbers we will mark the message as **legitimate**, otherwise we will check that they are not part of our email and phone blacklists. If there are any email or phone numbers on the blacklist we mark the message as **illegitimate**, otherwise we call our prediction model and return the result it gives us (**legitimate** or **illegitimate**).

4.2.3.3. PredictionModel

One of the most important elements will be our prediction model. Its objective will be to provide a tool that from the content of the SMS will allow us to differentiate if a message contains legitimate content or if unfortunately it contains illegitimate content. This class will be integrated as an object within the module that validates the content of the SMS (**SMSContentAnalyzer**).

In order to maintain the maximum degree of accuracy in the predictions, we will have a different model for each language we want to analyze. As a starting point, we will begin by supporting messages that come in **English** and **Spanish**, since practically all the messages we will receive will be in one of these two languages.

For each model, we will explain how they have been constructed, from the *creation of the dataset* to the *saving of the model*.

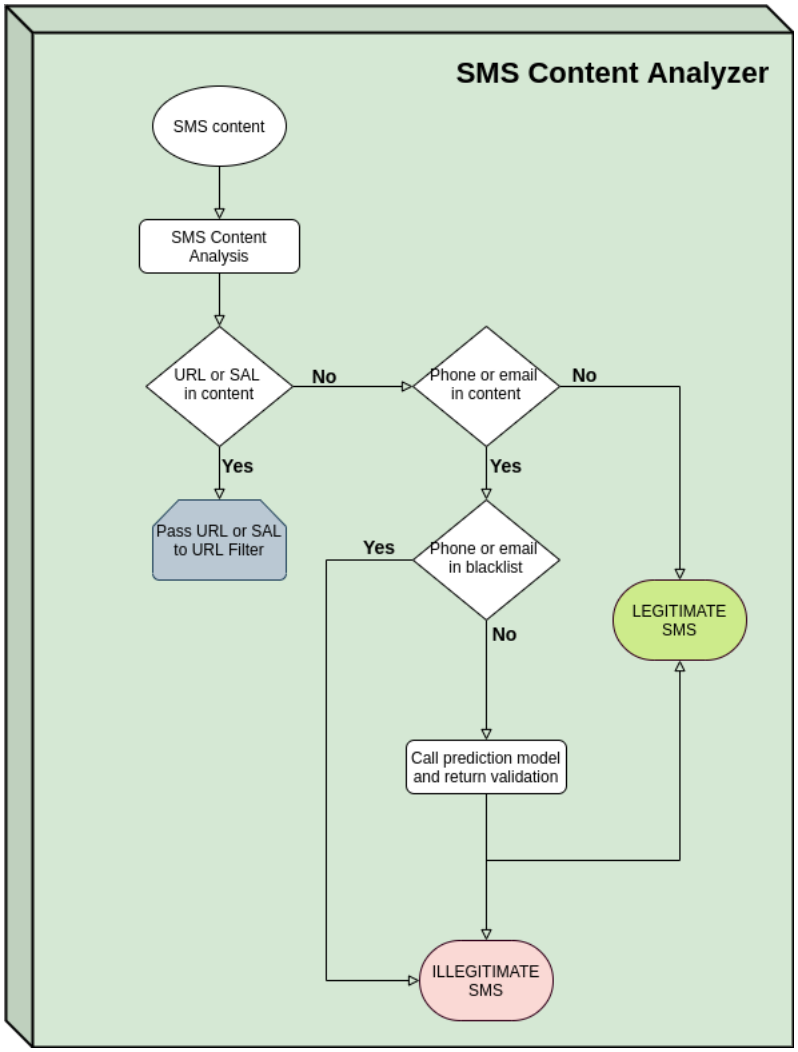

Figure 4.11.: SMS Content Analyzer flowchart

4.2.3.3.1. English model

To start we will use the dataset provided by T. Almeida [2] that contains 5,574 SMS, from that 4827 are ham[7] and the other 747 spam. Within the spam messages some are smishing and others not, this is because the smishing messages are a subset of spam messages. So, we have to discard from those spam messages, the ones that aren't smishing.

When we have filtered the spam messages from the smishing messages, we can see that the dataset contains less than a 5% of illegitimate messages (Consider illegitimate messages the ones that contains smishing inside and legitime the opposite case). To obtain a proportion of 90% of legitimate messages and 10% of illegitimate messages we will complement it with some smishing messages courtesy of Sandhya Mishra, author of [9].

For all the messages we will check some features about the messages like:

- **language**: If we're working with an english dataset, would be great to discard all messages in other language, i.e spanish messages.
- **linebreaks**: Some messages comes with multiple line breaks, to make dataset loading easier, we will use one line.

As we can see in Figure 4.12, the structure of the dataset will be: Once all of this pre filter

	label	text
0	legitimate	I think the other two still need to get cash b...
1	legitimate	Evry Emotion dsn't hav Words.Evry Wish dsn't h...
2	illegitimate	Todays Voda numbers ending 7548 are selected t...
3	legitimate	Wrong phone! This phone! I answer this one but...
4	illegitimate	Someone has conacted our dating service and en...
5	legitimate	We are supposed to meet to discuss abt our tri...
6	illegitimate	Had your mobile 11 months or more? U R entitle...
7	legitimate	Hey mr and I are going to the sea view and ha...
8	illegitimate	Do you want a NEW video phone750 anytime any n...
9	legitimate	Hey ! I want you ! I crave you ! I miss you ! ...

Figure 4.12.: English dataset structure

is done, we will combine all legitimate and illegitimate messages, do the shuffle of the rows and create the new combined dataset. In the Figures 4.13, 4.14 and 4.15 we can see the final proportions of the dataset as well as a **WordCloud**[8] of legitimate and illegitimate messages, which will allow us to see clearly which words are most used for each type of message.

[7] The term "ham" is currenty defined and understood to be a message or email generally desired and isn't considered spam.

[8] WordCloud is an electronic image that shows words used in a particular piece of electronic text or series of texts. The words are different sizes according to how often they are used in the text.

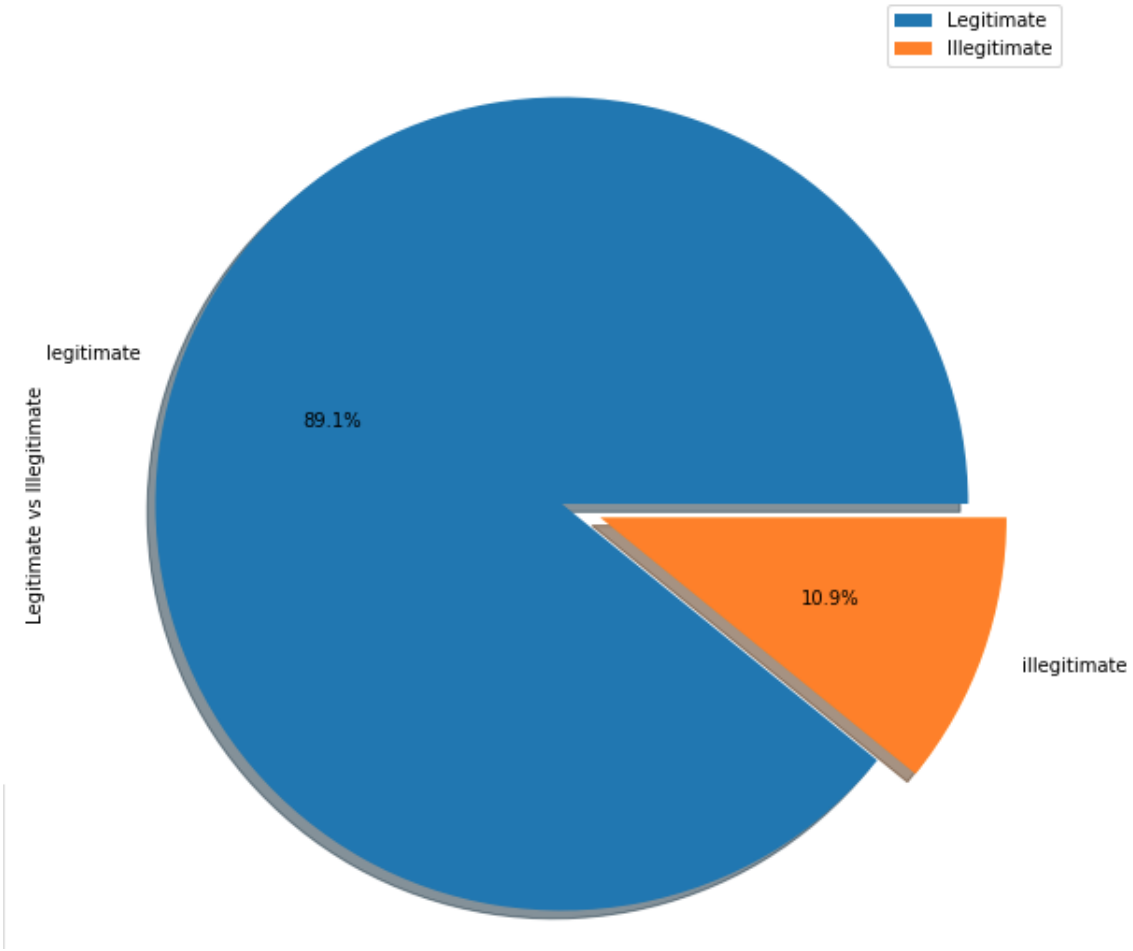

Figure 4.13.: English dataset pie graph disposition

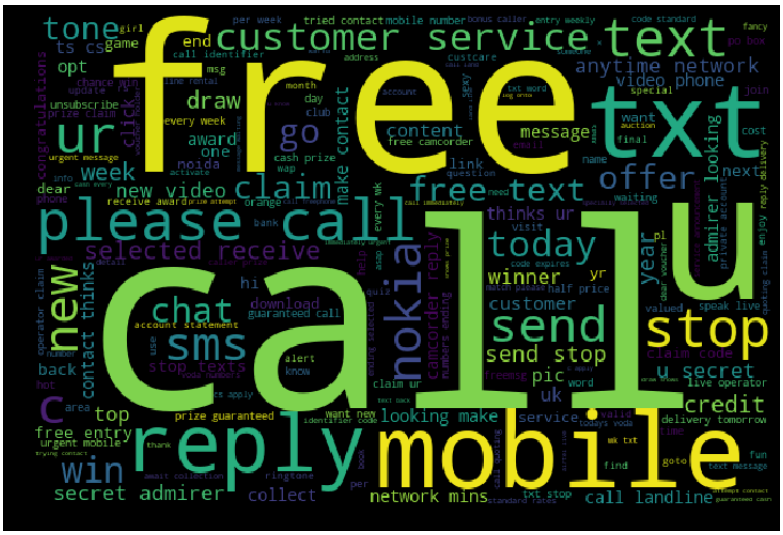

Figure 4.14.: English dataset illegitimate WordCloud

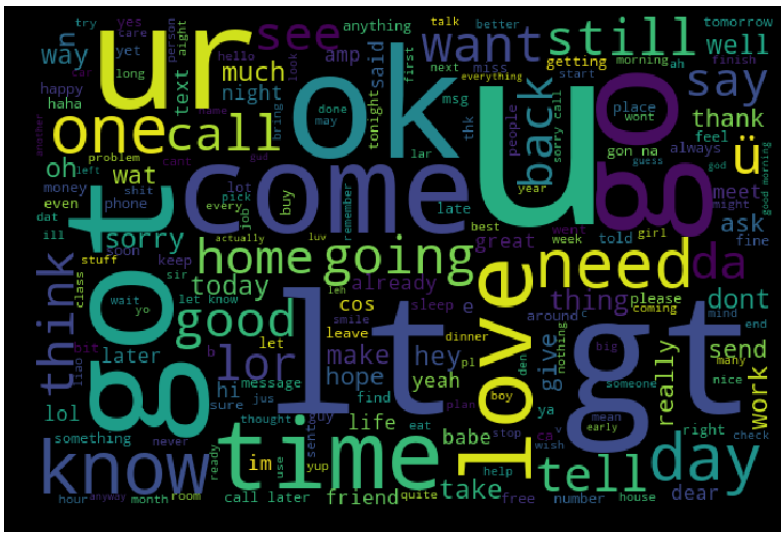

Figure 4.15.: English dataset legitimate WordCloud

Before training the model, we will do another data cleaning, removing punctuation symbols, stop words[9] and stemming[10] words.

Once this cleaning is done, we will convert the dataset into the **T**erm **F**requency **I**nverse **D**ocument **F**requency (TFIDF) representation, so that our machine learning model can understand the data. A TFIDF is a very common algorithm to transform text into a meaningful representation of numbers which is used to fit machine learning algorithms for predictions. So, for each message we'll extract some features containing the information about the text in a "readable" way for the machine learning algorithm.

The next step is to split the previous elaborated dataset, now in TFIDF representation, into training and testing, for our case we'll use the 80% of the total for training and the resting 20% for testing. For our project, we chose a Multinomial Naive Bayes model (See 2.2.1 for explanation) as in [9]. With this training data we'll fit the model to train it and with the testing data we'll do the corresponding predictions and save the outputs to extract the results later and print some interesting statistics like the accuraccy, precision and recall.

In order to make the model portable and usable on our system, we will save the vectoriser and model objects as **.sav** files. These extensions are usually used to store generic or data files, in our case it will contain all the necessary information so that we can load it as a Python object inside our code.

The final step will be to obtain the classification metrics to evaluate our prediction model.

[9] **Stop words** usually refers to the most common words in a language, there is no single universal list of stop words used by all natural language processing tools, and indeed not all tools even use such a list.

[10] **Stemming** is the process of reducing a word to its word stem that affixes to suffixes and prefixes or to the roots of words known as a lemma. Stemming is important in **N**atural **L**anguage **U**nderstanding (NLU) and NLP.

These are explained in the Annex A. The table 4.1 shows the results of the evaluation of

Class label	Precision	Recall	F1-Score	Support	Accuracy
0 (ILLEGITIMATE)	1.00	0.53	0.70	118	0.95
1 (LEGITIMATE)	0.95	1.0	0.97	965	

Table 4.1.: English prediction model classification metrics results

our Multinomial Naive Bayes model with the testing dataset. For **label 0 (illegitimate)**, we can see that the precision is 100 percent and the recall 53 percent. This tells us that when the model predicts illegitimate, it is right in 100 percent of the cases, but when the message is illegitimate, it predicts legitimate in 47 percent of the cases. The F1-Score metric is the harmonic weighting between accuracy and recall, in this case with a value of 0.70. The support simply tells us the number of occurrences of a class in the dataset.

For **label 1 (legitimate)**, we can observe that despite having obtained a low recall for class 0, the precision is 95 percent, this is due to the huge amount of legitimate messages (965) and the low amount of illegitimate messages (118) in the dataset. In this case the recall is 100 percent since there are no legitimate messages that are predicted to be illegitimate.

To conclude we can see that the accuracy is very high, since for legitimate messages practically all of them are predicted correctly, the percentage drops when for illegitimate messages some are predicted as legitimate, i.e. our model is quite permissive when categorising illegitimate messages.

4.2.3.3.2. Spanish model

As we have a database where all SMS messages are stored in the company, we will collect as many messages as we can from which there is prior authorization to use them. After a large filtering of messages, we have found a large number of legitimate messages, but no illegitimate messages (containing smishing). After an exhaustive search on the Internet, hardly any illegitimate messages were found in Spanish. To solve this issue, we have translated all the illegitimate messages of the dataset in English to Spanish using the library **googletrans** and instantiating the object **Translator**, by doing it through an script we could use it in the future if we wanted to add more languages and we didn't find any data.

As we can see in Figure 4.16, the dataset structure is the same as in the English model. Once we have filtered all data, we will combine all legitimate and illegitimate message, do the shuffle of the rows and create the new combined dataset, as well as explained in English model. The obtained pie graph and the corrresponding WordCloud representation are shown in Figures 4.17, 4.18 and 4.19 respectively.

Once we have created our dataset, the next steps will be te same as in the English model with no difference. So, we pass to comment the classification metrics table: For **label 0 (illegitimate)**, we can see that the precision is 100 percent while the recall drops to 51 percent, and consequently its F1-Score. This class had 118 elements in the testing dataset.

	label	text
0	legitimate	Ahora no puedo hablar. ¨Qu, pasa?
1	legitimate	LE RECORDAMOS SU PROXIMA CITA DE PELUQUERIA 0...
2	legitimate	Bienvenido a Viasat, `Como fue tu instalacion?...
3	illegitimate	Fantasy Football está de vuelta en tu televiso...
4	legitimate	Necesito el numero de telefono completo por favor
5	legitimate	Esta es tu clave para la renovacion de contrat...
6	legitimate	Con abuelo
7	legitimate	GENERALI - En cualquier momento puede consulta...
8	legitimate	Exacto, se ha renovado el plan de 24GB+Llamada...
9	legitimate	Introduce este código 350-435 para confirmar t...

Figure 4.16.: Spanish dataset structure

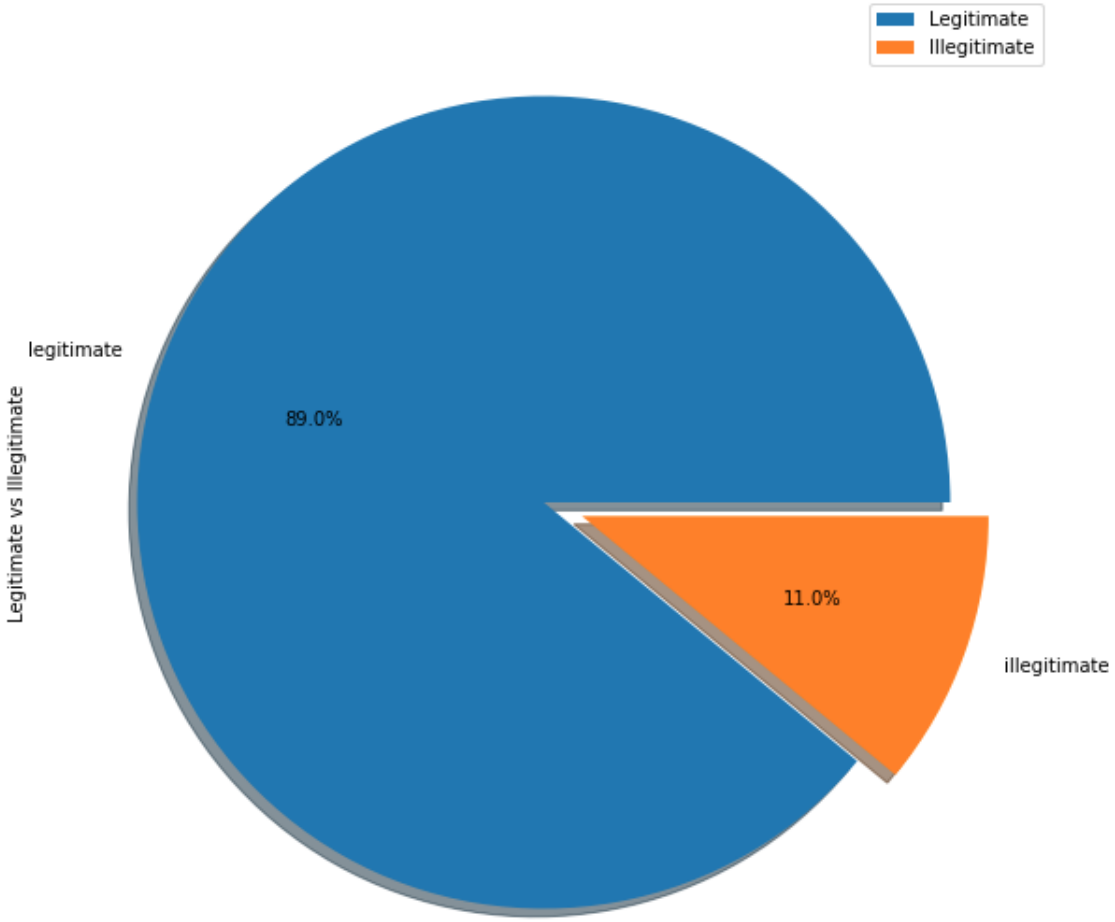

Figure 4.17.: Spanish dataset pie graph disposition

Figure 4.18.: Spanish dataset illegitimate WordCloud

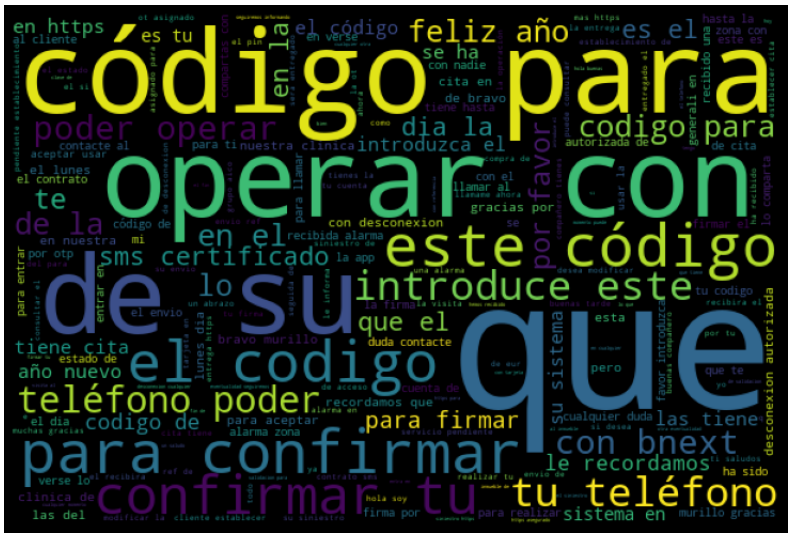

Figure 4.19.: Spanish dataset legitimate WordCloud

Class label	Precision	Recall	F1-Score	Support	Accuracy
0 (illegitimate)	1.00	0.51	0.67	118	0.95
1 (legitimate)	0.94	1.00	0.97	953	

Table 4.2.: Spanish prediction model classification metrics results

For **label 1 (legitimate)**, we can see that the accuracy is 94 percent and its recall is 100 percent, taking into account that this class has 953 samples in the testing dataset.

The accuracy obtained is high, even though variations in the illegitimate messages are also observed, conditioned by exactly the same causes as in the previous model.

4.2.3.4. Blacklist

This class will interact with the corresponding database tables and its objective is to bring to the other classes the abstraction to do the following operations:

- **add**: Add an element to the source blacklist.
- **contains**: Check if an element is contained within the source blacklist.
- **remove**: Remove an specific element from the source blacklist.

4.2.3.5. URLFilter

This class is the **second layer** that will perform a validation of the contained URLs in the SMS text. The flowchart is represented in Figure 4.20.

The validation process performed is the same for every URL detected in the SMS text. The validation starts converting the long URL to short URL, this means following redirections if has. If the short URL is within the URL Blacklist we'll mark the message as **ILLEGITIMATE**, otherwise we'll check some relevant features about the URL.

The features to check are the following 4:

1. Check if the age of domain is lower than 6 months
2. Check if the short URL contains the at(@) tag
3. Check if the short URL contains the hyphen(-) character
4. Check if the short URL contains more than 5 dots (.)

If 3 or more conditions are achieved, we'll assume that the URL is fraudulent and consequently mark the message as **ILLEGITIMATE**, otherwise we'll pass the source code of the URL to the **Source Code Analyzer**.

4.2.3.6. SourceCodeAnalyzer

This class is the **third layer** that will perform a validation of the URLs source code contained in the SMS text. The flowchart is represented in Figure 4.21.

The validation starts checking if there is any form tag in the source code, if it's the case we'll check that the domain URL of the form tag is the same as the URL domain, if it's

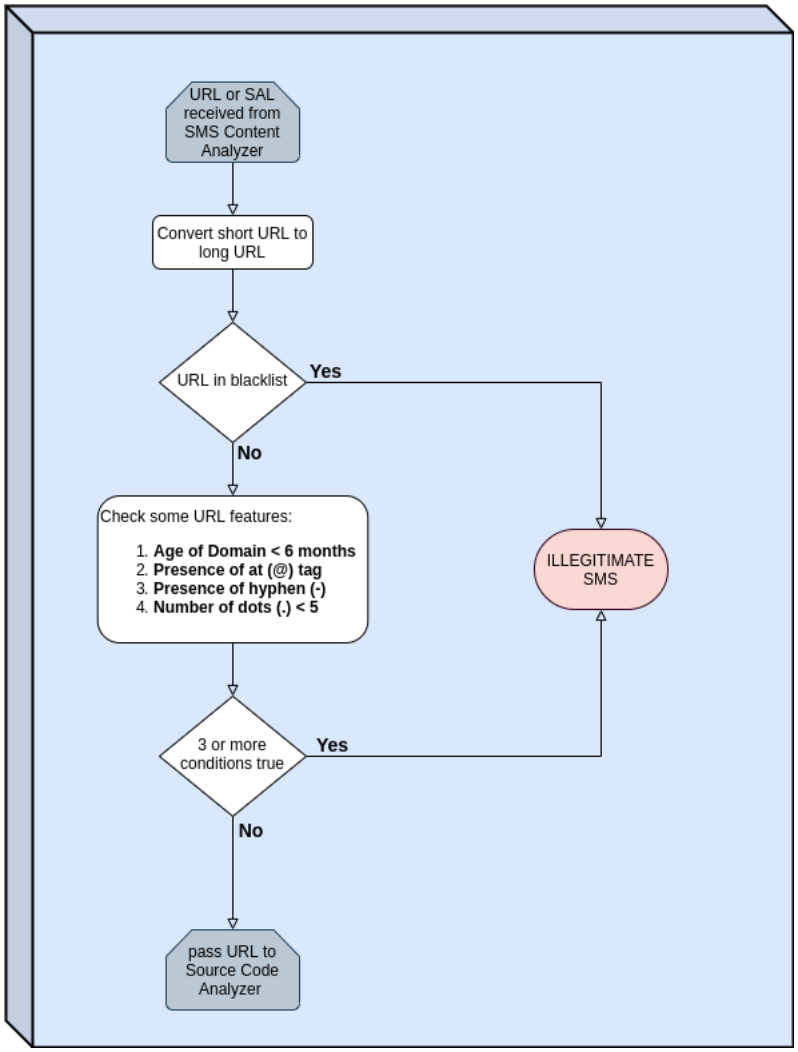

Figure 4.20.: URL Filter flowchart

different we'll mark the SMS as **ILLEGITIMATE**.

If any form tag is detected on the source code, or the form tag URL domain are the same as the URL domain, we'll skip to the next and final validation module, the **APK Download Detector**.

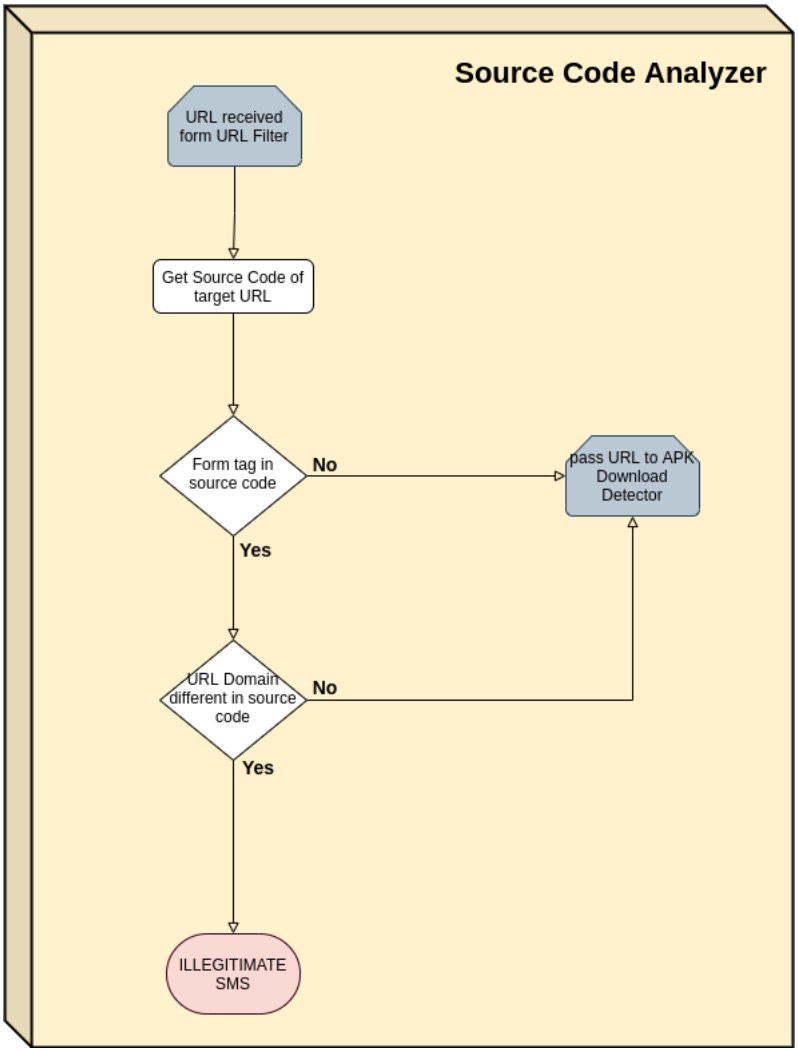

Figure 4.21.: Source Code Analyzer flowchart

4.2.3.7. APKDownloadDetector

This class is the **fourth layer** that will perform a validation to check if the URLs contains a downloadable apk file. The flowchart is represented in Figure 4.22.

The validation starts launching a recursive function to check if in the headers there's a downloadable apk file, if we detect a downloadable apk file we'll mark the message as **ILLE-**

GITIMATE, otherwise we'll look the source code for possible iframe's[11] because if opened from a browser, the user could be redirected to a malicious page. For each iframe detected in the source code we'll apply the same recursive validation function in order to catch those possible malicious redirections.

To avoid possible infinite loops, we have implemented a cut index, initially set to 5. This cut index indicates the maximum level of depth that we want to cover.

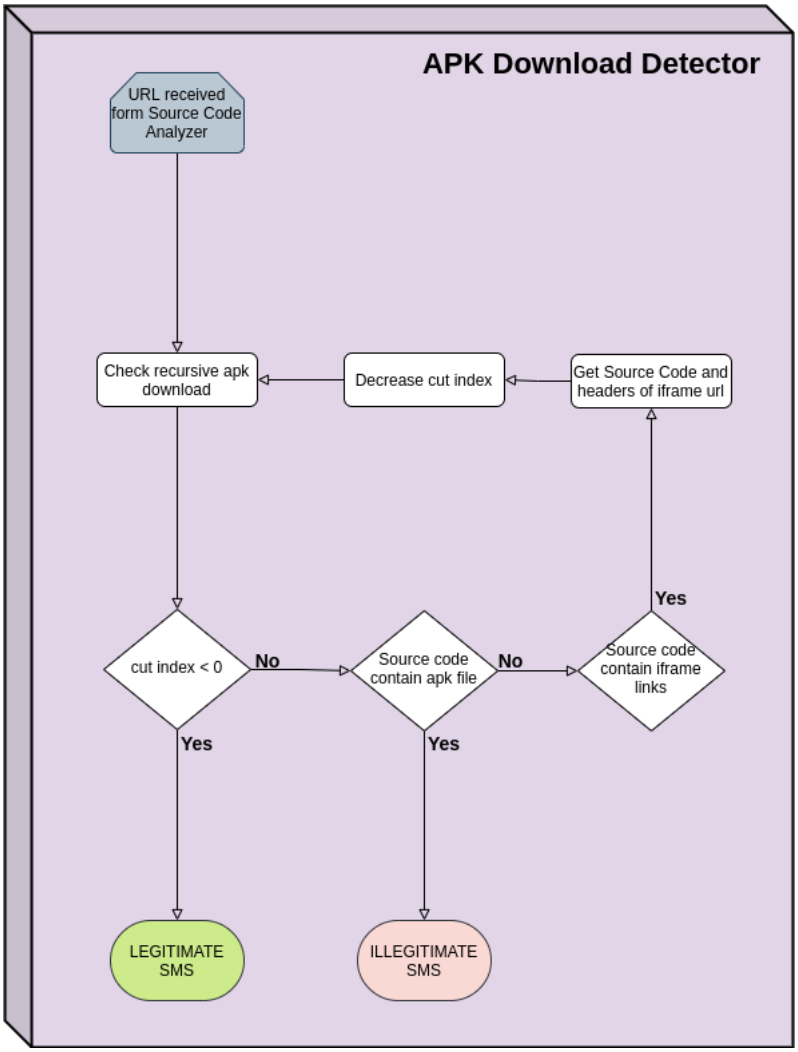

Figure 4.22.: APK Download Detector flowchart

[11] An iframe (short for inline frame) is an HTML element that allows an external webpage to be embedded in an HTML document. Unlike traditional frames, which were used to create the structure of a webpage, iframes can be inserted anywhere within a webpage layout.

4.2.4. Database

For this project we have chosen a relational MYSQL database, since it is one of the best database managers due to its ease of installation, administration and low cost. The project doesn't requires a complex and costly database, so it is not necessary to overcomplicate our lives looking for a paid database engine.

As we can see, the main scheme is quite simple, it mainly has 4 tables:

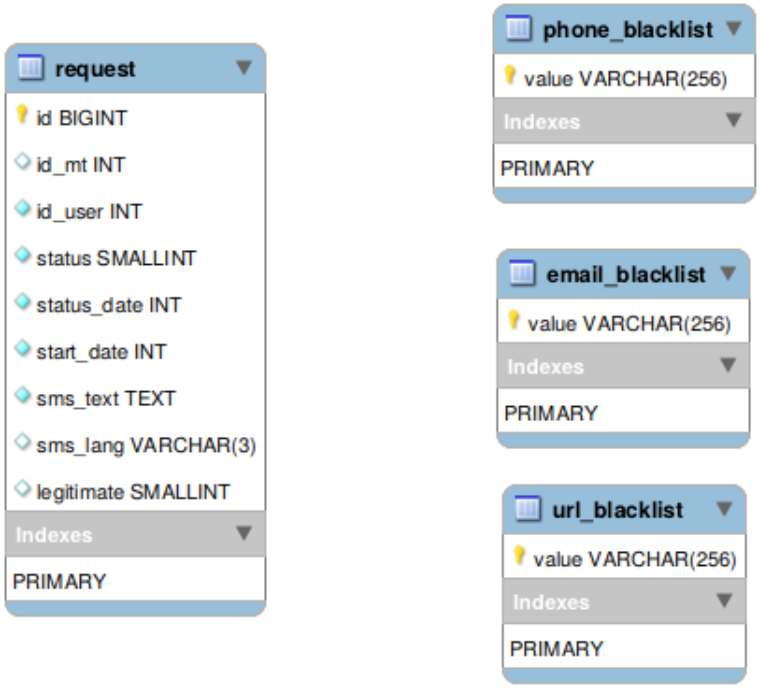

Figure 4.23.: Smishing UML

1. **request**: For each new request it will contain all the necessary information for validation and retrieval of the validation results.

2. **phone_blacklist**: It will act as a box where we will add or remove items from the phone blacklist.

3. **email_blacklist**: It will act as a box where we will add or remove items from the email blacklist.

4. **url_blacklist**: It will act as a box where we will add or remove items from the URL blacklist.

The script to create the schema and the tables is described in Annex B.

4.2.5. Local Storage

In order to save both the system logs and the objects of the prediction models, we could use any file system. For a first local version of the system we have used the file system that comes by default in our machine, in this case **ext4**[12].

```
/dev/sda2       ext4       234G   184G   38G  83% /
```

4.2.6. Modules

4.2.6.1. Database

This class will be in charge of providing all the necessary methods to the rest of the classes, so that they can perform their operations. The main implementation of this class consists of the following methods:

- **insert_request**: Insert the necessary data for validation into the request table to the database.
- **select_request**: Select the necessary data to retrieve data from the database.
- **update_request_status**: Update request status and status date for an specific id.
- **update_request_done**: Update request status to STATUS_DONE, status date, end date and legitimate for an specific id.
- **insert_to_blacklist**: Insert value to the corresponding blacklist.
- **select_from_blacklist**: Select value from the corresponding blacklist.
- **delete_from_blacklist**: Delete value from the corresponding blacklist.

4.2.6.2. Logger

This class will be in charge of providing an object, which will be instantiated by each class interested in logging. In this way we will have a record of everything that happens in the code and it will help us in case of possible errors or improvements. For the log complexity we will mainly define 5 types of log levels, ranked from lowest to highest priority, extracted from Logger Python facilities[13]:

- **debug**: Detailed information, typically of interest only when diagnosing problems.
- **info**: Confirmation that things are working as expected.
- **warning**: An indication that something unexpected happened, or indicative of some problem in the near future (e.g. 'disk space low'). The software is still working as expected.

[12] It is a log file system that was conceived as a compatible enhancement to ext3.
[13] https://docs.python.org/3/library/logging.html

- **error**: Due to a more serious problem, the software has not been able to perform some function.
- **critical**: A serious error, indicating that the program itself may be unable to continue running.

The format defined to log is the following:

```
logging.Formatter("%(asctime)s,%(msecs)d %(levelname)s {0} {1}:%(message)s".format(uuid.uuid4().hex, name))
```

So the output will look like in Figure 4.24

```
2021-05-03 07:57:26,874,874 DEBUG d61c53d1cc1a4bde9f71e9fb39435d45 ClassToLog.py:118 time:1620021446 :main msg: this is a debug log!
2021-05-03 07:57:26,874,874 INFO d61c53d1cc1a4bde9f71e9fb39435d45 ClassToLog.py:119 time:1620021446 :main msg: this is an info log!
2021-05-03 07:57:26,874,874 WARNING d61c53d1cc1a4bde9f71e9fb39435d45 ClassToLog.py:120 time:1620021446 :main msg: this is a warning log!
2021-05-03 07:57:26,874,874 ERROR d61c53d1cc1a4bde9f71e9fb39435d45 ClassToLog.py:121 time:1620021446 :main msg: this is an error log!
2021-05-03 07:57:26,874,874 CRITICAL d61c53d1cc1a4bde9f71e9fb39435d45 ClassToLog.py:122 time:1620021446 :main msg: this is a critical log!
```

Figure 4.24.: Log output format example

4.2.6.3. module

This file will contain all the generic functions and imports that will be common in the different core and API classes. That's why to make the code easier and cleaner each class will import this file and consequently import the rest of the stuff in it, this includes both count files, libraries and generic methods.

4.3. Evaluation

4.3.1. Testing

As we seen in [8], at a high level, we need to differentiate between *manual* and *automated* tests. **Manual testing** is done in person, by clicking through the application or interacting with the software and APIs with the appropriate tools. This is very expensive as it requires someone to set up an environment and execute the tests manually, and it can be prone to human errors as the tester might make typos or omit steps in the test script.

Automated tests, on the other hand, are performed by a machine that executes a test script that has been written in advance. These tests can vary a lot in complexity, from checking a single method in a class to making sure that performing a sequence of complex actions in the UI leads to the same results. It's much more robust and reliable than manual tests - but the quality of your automated tests depends on how well your test scripts have been written.

The type of tests we will perform will be **Automated tests**, since we do not have a graphical interface to interact with the product, it does not make much sense. We are also interested in being able to perform continuous integration and continuous delivery because it's a great way to scale the **Q**uality **A**ssurance (QA) process as we add new features to our application.

For our case, we will mainly use three well known types of tests, *unit tests, integration tests and functional tests*. We will not separate the classes of tests by type, we are interested in being able to test each class individually and therefore each class can contain different types of tests. In the following we will briefly explain what the main tests we will use are and finally we will show the results by running the pytest tool.

4.3.1.1. Unitary tests

Unit tests are very low level, close to the source of your application. They consist in testing individual methods and functions of the classes, components or modules used by your software. Unit tests are in general quite cheap to automate and can be run very quickly by a continuous integration server.

4.3.1.2. Integration tests

Integration tests verify that different modules or services used by your application work well together. For example, it can be testing the interaction with the database or making sure that microservices work together as expected. These types of tests are more expensive to run as they require multiple parts of the application to be up and running.

4.3.1.3. Functional tests

Functional tests focus on the business requirements of an application. They only verify the output of an action and do not check the intermediate states of the system when performing that action.

There is sometimes a confusion between integration tests and functional tests as they both require multiple components to interact with each other. The difference is that an integration test may simply verify that you can query the database while a functional test would expect to get a specific value from the database as defined by the product requirements.

4.3.1.4. Results

Below are all the evidences and results generated from the tests performed on the modules that make up the system. All tests have been executed using the pytest tool, which has allowed us to implement all the types of tests mentioned above and automate them to promote continuous integration.

As mentioned above, each test may contain different types of tests (unitary, integration and functional). For the compilation of the evidence, each test has been executed individually to provide greater clarity, but if we do not specify the test file and only launch the pytest command inside the corresponding folder, it will unify all the test files that exist and launch them within the same execution pipeline.

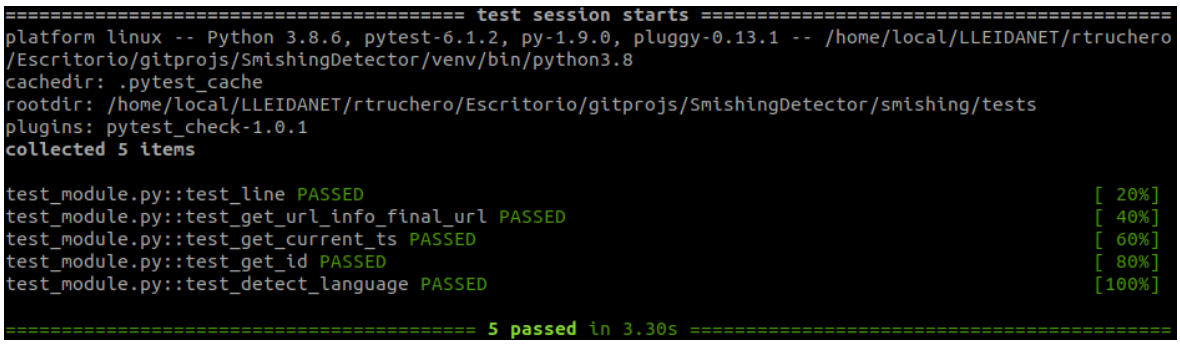

Figure 4.25.: Test module evidence

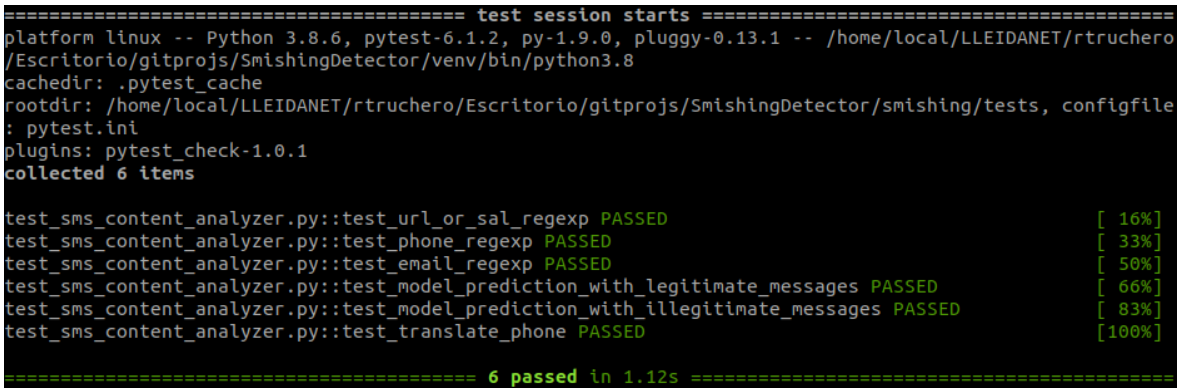

Figure 4.26.: Test SMS Content Analyzer evidence

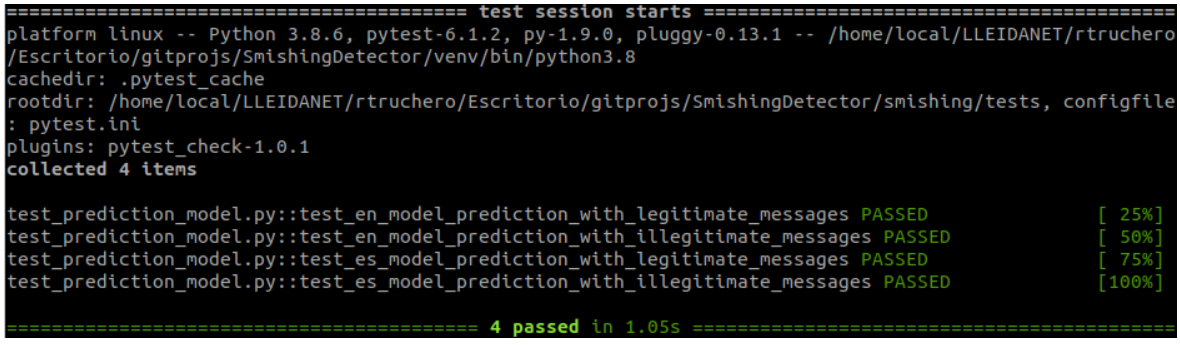

Figure 4.27.: Test Prediction Model evidence

```
========================================= test session starts ==========================================
platform linux -- Python 3.8.6, pytest-6.1.2, py-1.9.0, pluggy-0.13.1 -- /home/local/LLEIDANET/rtruchero
/Escritorio/gitprojs/SmishingDetector/venv/bin/python3.8
cachedir: .pytest_cache
rootdir: /home/local/LLEIDANET/rtruchero/Escritorio/gitprojs/SmishingDetector/smishing/tests, configfile
: pytest.ini
plugins: pytest_check-1.0.1
collected 7 items

test_url_filter.py::test_unquote PASSED                                                          [ 14%]
test_url_filter.py::test_check_age_of_domain PASSED                                              [ 28%]
test_url_filter.py::test_check_presence_of_at_tag PASSED                                         [ 42%]
test_url_filter.py::test_check_presence_of_hyphen PASSED                                         [ 57%]
test_url_filter.py::test_check_number_of_dots PASSED                                             [ 71%]
test_url_filter.py::test_check_url PASSED                                                        [ 85%]
test_url_filter.py::test_validate_illegitimate PASSED                                            [100%]

========================================== 7 passed in 1.58s ===========================================
```

Figure 4.28.: Test URL Filter evidence

```
========================================= test session starts ==========================================
platform linux -- Python 3.8.6, pytest-6.1.2, py-1.9.0, pluggy-0.13.1 -- /home/local/LLEIDANET/rtruchero
/Escritorio/gitprojs/SmishingDetector/venv/bin/python3.8
cachedir: .pytest_cache
rootdir: /home/local/LLEIDANET/rtruchero/Escritorio/gitprojs/SmishingDetector/smishing/tests, configfile
: pytest.ini
plugins: pytest_check-1.0.1
collected 7 items

test_source_code_analyzer.py::test_valid_absolute_urls PASSED                                    [ 14%]
test_source_code_analyzer.py::test_invalid_absolute_urls PASSED                                  [ 28%]
test_source_code_analyzer.py::test_get_action_domain PASSED                                      [ 42%]
test_source_code_analyzer.py::test_empty_html_form_tags_is_valid_domain_form_tags PASSED         [ 57%]
test_source_code_analyzer.py::test_xml_file_is_valid_domain_form_tags PASSED                     [ 71%]
test_source_code_analyzer.py::test_local_html_action_is_valid_domain_form_tags PASSED            [ 85%]
test_source_code_analyzer.py::test_different_html_action_url_is_invalid_domain_form_tags PASSED  [100%]

========================================== 7 passed in 1.05s ===========================================
```

Figure 4.29.: Test Source Code Analyzer evidence

```
========================================= test session starts ==========================================
platform linux -- Python 3.8.6, pytest-6.1.2, py-1.9.0, pluggy-0.13.1 -- /home/local/LLEIDANET/rtruchero
/Escritorio/gitprojs/SmishingDetector/venv/bin/python3.8
cachedir: .pytest_cache
rootdir: /home/local/LLEIDANET/rtruchero/Escritorio/gitprojs/SmishingDetector/smishing/tests, configfile
: pytest.ini
plugins: pytest_check-1.0.1
collected 3 items

test_apk_download_detector.py::test_apk_is_downloaded PASSED                                     [ 33%]
test_apk_download_detector.py::test_contains_apk PASSED                                          [ 66%]
test_apk_download_detector.py::test_not_contains_apk PASSED                                      [100%]

========================================== 3 passed in 0.34s ===========================================
```

Figure 4.30.: Test APK Download Detector evidence

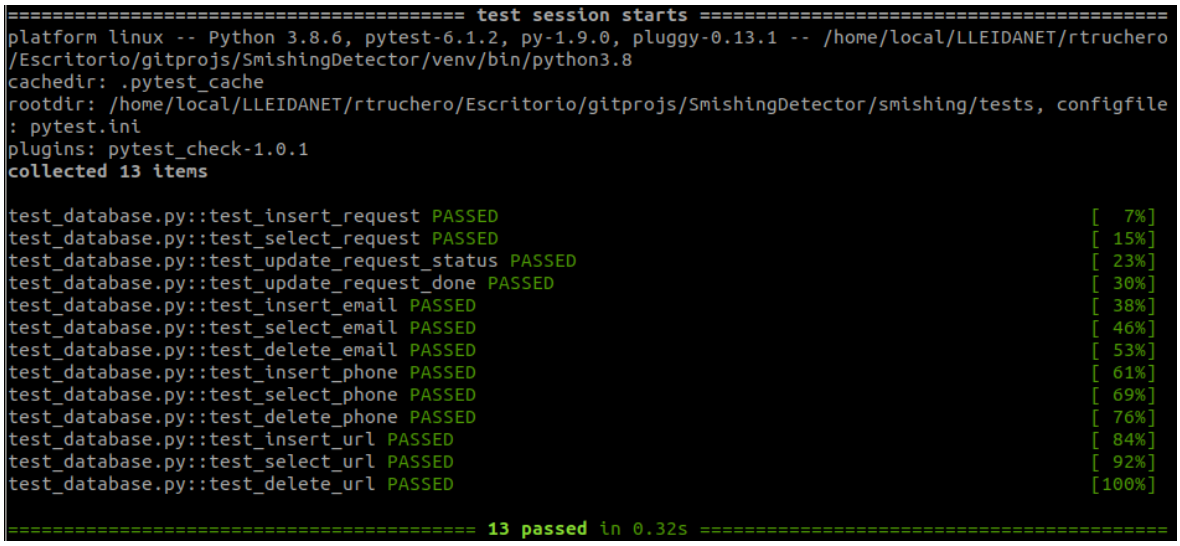

Figure 4.31.: Test Database evidence

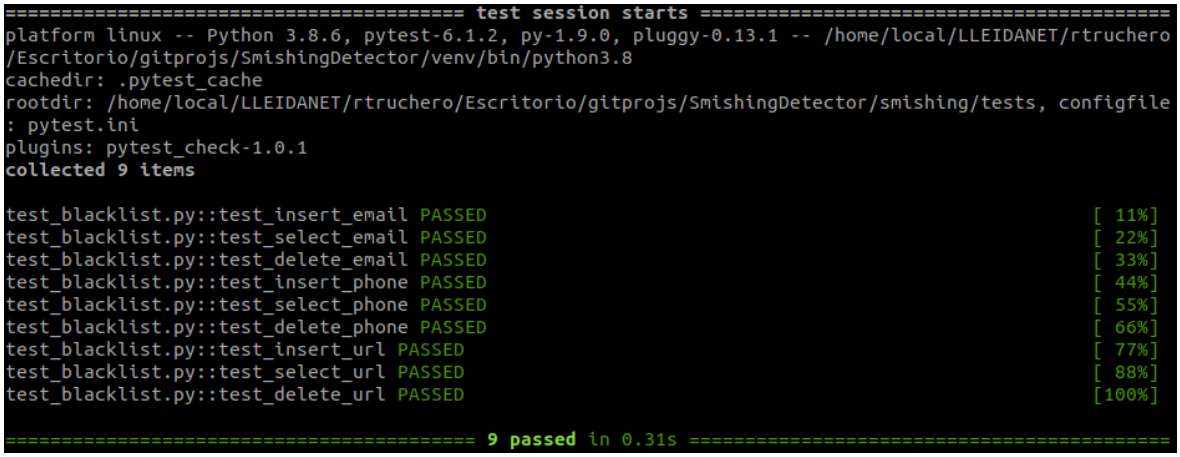

Figure 4.32.: Test Blacklist evidence

```
========================================= test session starts ==========================================
platform linux -- Python 3.8.6, pytest-6.1.2, py-1.9.0, pluggy-0.13.1 -- /home/local/LLEIDANET/rtruchero
/Escritorio/gitprojs/SmishingDetector/venv/bin/python3.8
cachedir: .pytest_cache
rootdir: /home/local/LLEIDANET/rtruchero/Escritorio/gitprojs/SmishingDetector/smishing/tests, configfile
: pytest.ini
plugins: pytest_check-1.0.1
collected 13 items

test_request.py::test_send_json_error PASSED                                                    [  7%]
test_request.py::test_send_json_kwargs PASSED                                                   [ 15%]
test_request.py::test_send_id PASSED                                                            [ 23%]
test_request.py::test_validate_ok PASSED                                                        [ 30%]
test_request.py::test_validate_non_json_request PASSED                                          [ 38%]
test_request.py::test_validate_invalid_id_mt_field PASSED                                       [ 46%]
test_request.py::test_validate_invalid_id_user_field PASSED                                     [ 53%]
test_request.py::test_validate_invalid_text_field PASSED                                        [ 61%]
test_request.py::test_get_insert_values PASSED                                                  [ 69%]
test_request.py::test_results_ok PASSED                                                         [ 76%]
test_request.py::test_results_without_id PASSED                                                 [ 84%]
test_request.py::test_results_with_non_numeric_id PASSED                                        [ 92%]
test_request.py::test_results_request_non_existing_id PASSED                                    [100%]

========================================== 13 passed in 0.88s ==========================================
```

Figure 4.33.: Test Request evidence

4.3.2. Profiling

This section is focused on analyzing the parts that generate a higher time penalization to our system using profiling tools, in this case Python's **cProfile**. For this profiling we will focus on the Core of the application, that is, once we have received the message to be analyzed, which parts are critical in the validation of the result. For this we will create a Profiler class, which will make 10 calls to an endpoint previously prepared locally. This endpoint contains an .xapk file with .apk files inside, just to make sure that the system is executed until the last module, that is to say, we put ourselves in the worst possible case that would be to have to make the longest and most expensive flow of validations.

From the results of running the profiler we will generate two files:

1. Profiling results ordered by the number of calls
2. Profiling results ordered by the internal time

Since the goal is extract critical parts in the code to optimize the service time as much as possible, we will look at the second file, which will contain the methods that consume the most time during execution. We will not look too much at the first one, since the number of times a function is called is not strictly related to its execution time, and it is not exactly what we are looking for.

As we can see in the figure 4.34, there is a critical flow of operations, these are all related to HTTP GET requests, spending more than 87 seconds to read result in 20 requests. These requests are made to validate the content of the URL's (source code, response headers, apk file downloads, etc.). Therefore, at a certain point we depend on the other side to receive a quick response.

sync_output_calls.txt

SmishingDetector > smishing > localstore > cprofile > sync_output_calls.txt

```
         595446 function calls (594232 primitive calls) in 94.723 seconds

   Ordered by: call count

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    44986    0.014    0.000    0.014    0.000 /usr/lib/python3.8/ssl.py:1078(_checkClosed)
39997/39806    0.018    0.000    0.019    0.000 {built-in method builtins.len}
    24150    0.005    0.000    0.005    0.000 {method 'decode' of 'bytes' objects}
    22483    0.017    0.000    0.017    0.000 /usr/lib/python3.8/socket.py:694(readable)
    22463    0.018    0.000    0.018    0.000 {method '_checkClosed' of '_io._IOBase' objects}
    22463    0.025    0.000    0.041    0.000 {method '_checkReadable' of '_io._IOBase' objects}
    22463    0.083    0.000   87.327    0.004 /usr/lib/python3.8/socket.py:655(readinto)
    22463   87.030    0.004   87.030    0.004 {method 'read' of '_ssl._SSLSocket' objects}
    22463    0.052    0.000   87.089    0.004 /usr/lib/python3.8/ssl.py:1090(read)
    22463    0.079    0.000   87.185    0.004 /usr/lib/python3.8/ssl.py:1230(recv_into)
19720/19320    0.005    0.000    0.007    0.000 {built-in method builtins.isinstance}
18096/18036    0.005    0.000    0.006    0.000 {method 'encode' of 'str' objects}
    16880    0.005    0.000    0.009    0.000 /home/local/LLEIDANET/rtruchero/Escritorio/gitprojs/SmishingDetector/venv/lib/python3.8/os.py:752(decode)
    12474    0.119    0.000    4.167    0.000 {method 'read' of '_io.BufferedReader' objects}
    12466    0.029    0.000    4.203    0.000 /usr/lib/python3.8/http/client.py:602(_safe_read)
    12298    0.009    0.000    0.020    0.000 /home/local/LLEIDANET/rtruchero/Escritorio/gitprojs/SmishingDetector/venv/lib/python3.8/site-packages/urllib3/response.py:689(_update_chunk_length)
    12278    0.010    0.000    0.011    0.000 /home/local/LLEIDANET/rtruchero/Escritorio/gitprojs/SmishingDetector/venv/lib/python3.8/site-packages/urllib3/response.py:392(_decode)
    12278    0.042    0.000    4.245    0.000 /home/local/LLEIDANET/rtruchero/Escritorio/gitprojs/SmishingDetector/venv/lib/python3.8/site-packages/urllib3/response.py:703(_handle_chunk)
```

sync_output_time.txt

SmishingDetector > smishing > localstore > cprofile > sync_output_time.txt

```
         595446 function calls (594232 primitive calls) in 94.723 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    22463   87.030    0.004   87.030    0.004 {method 'read' of '_ssl._SSLSocket' objects}
       40    2.805    0.070    2.805    0.070 {method 'connect' of '_socket.socket' objects}
       40    2.461    0.062    2.461    0.062 {method 'recv' of '_socket.socket' objects}
       20    0.871    0.044    0.871    0.044 {method 'do_handshake' of '_ssl._SSLSocket' objects}
       20    0.305    0.015    0.305    0.015 {method 'load_verify_locations' of '_ssl._SSLContext' objects}
    12474    0.119    0.000    4.167    0.000 {method 'read' of '_io.BufferedReader' objects}
      210    0.109    0.001    4.467    0.021 {method 'join' of 'bytes' objects}
       20    0.092    0.005    0.092    0.005 {built-in method _socket.getaddrinfo}
    22463    0.083    0.000   87.327    0.004 /usr/lib/python3.8/socket.py:655(readinto)
    22463    0.079    0.000   87.185    0.004 /usr/lib/python3.8/ssl.py:1230(recv_into)
      200    0.074    0.000    0.074    0.000 {method 'findall' of 're.Pattern' objects}
    12180    0.057    0.000    4.335    0.000 /home/local/LLEIDANET/rtruchero/Escritorio/gitprojs/SmishingDetector/venv/lib/python3.8/site-packages/urllib3/response.py:725(read_chunked)
    22463    0.052    0.000   87.089    0.004 /usr/lib/python3.8/ssl.py:1090(read)
    12278    0.042    0.000    4.245    0.000 /home/local/LLEIDANET/rtruchero/Escritorio/gitprojs/SmishingDetector/venv/lib/python3.8/site-packages/urllib3/response.py:703(_handle_chunk)
    12466    0.029    0.000    4.203    0.000 /usr/lib/python3.8/http/client.py:602(_safe_read)
    22463    0.025    0.000    0.041    0.000 {method '_checkReadable' of '_io._IOBase' objects}
    22463    0.018    0.000    0.018    0.000 {method '_checkClosed' of '_io._IOBase' objects}
39997/39806    0.018    0.000    0.019    0.000 {built-in method builtins.len}
```

Figure 4.34.: Profiling calls and time output

The rest of the operations have a penalty of less than 0.074 seconds when launching 20 requests, if we look at it for each request it does not even reach 1 millisecond. Therefore, it is not worth modifying the code further. So, we will try to optimize as much as we can the **H**yper**T**ext **T**ransfer **P**rotocol (HTTP) communication to gain a few milliseconds to the many requests.

4.3.3. Optimization

In this part we will apply the possible corrections to the critical points extracted from the profiling, ensuring that we do not break anything and that the system continues working correctly. This last point is why automatic testing in an application is so important, it helps us to save time and minimize human errors when testing, ensuring that everything we had previously tested continues working after each change.

The main bottle neck of our system was the HTTP requests so, in order to mitigate or palliate this problem, two possible new implementations will be analyzed:

1. **Asynchronous HTTP Requests**: We'll use the **aiohttp** and **asyncio** libraries, to create tasks, each one of those tasks will request the corrresponding URL, and once it finished will send back the information to the event loop.
2. **Threaded HTTP Requests**: We'll use a library that uses requests called **requests_features** which uses threads.

After the implementation of these two new strategies, the tests have been proposed according to the following parameters:

- **Casuistry**:
  - **Best case**: The first URL to check is a malicious one, so there's no need to check all URLs to mark the message as **ILLEGITIMATE**.
  - **Worst case**: All URLs are **LEGITIMATE** excepts the last one, so we need to check all the URLs before marking the message as **ILLEGITIMATE**.
- **Number of URLs contained in the SMS**: 1, 2, 4 or 8.
- **Strategies**:
  - Synchronous HTTP Requests
  - Asynchronous HTTP Requests
  - Threaded HTTP Requests

Also is interesting to show the machine where the tests will run. The main features are shown in Figure 4.35.

After the implementation of each new strategy and some tests, we extracted the results in terms of total time and total calls, divided in 2 tables, one for the best casuistry (Table 4.3)

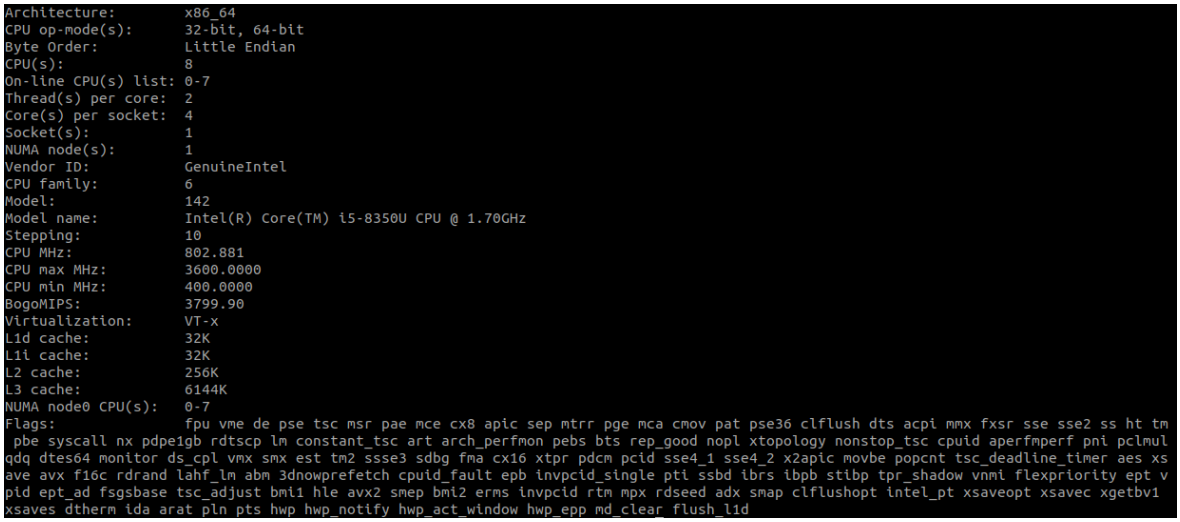

Figure 4.35.: Local Machine CPU architecture

Strategy	Total time (s)				Total calls			
	1	2	4	8	1	2	4	8
Synchronous HTTP requests	10,054	9,981	10,417	9,938	93060	53778	53785	53746
Asynchronous HTTP Requests	10,077	9,506	9,988	9,106	46215	10638	17468	30522
Threaded HTTP Requests	8,713	10,508	10,413	10,909	4274	4348	4508	4816

Table 4.3.: Best case casuistry comparison in relation to the number of URLs (1,2,4,8)

Strategy	Total time (s)				Total calls			
	1	2	4	8	1	2	4	8
Synchronous HTTP requests	9,696	9,605	12,539	16,19	93049	68021	93379	146059
Asynchronous HTTP Requests	6,931	9,85	9,351	14,84	7304	15894	33154	66763
Threaded HTTP Requests	10,342	10,126	10,904	13,325	43162	4478	12522	25802

Table 4.4.: Worst case casuistry comparison in relation to the number of URLs (1,2,4,8)

and other for the worst casuistry (Table 4.4). All the results are extracted from the outpout file of the **cProfiler** tool.

In table 4.3, where the results in the **best case** are shown, we can see that the total times are very similar, since only one request is made for all cases, that is why the strategy with threads is affected in comparison to the others. The strategy that gets the best performance is the *asynchronous version*, except for the total number of calls, which is better with the *threads version*.

If we look at table 4.4, where the results of the **worst case** are shown, here the differences can be better appreciated. We have that for the *synchronous version* the total time consumed when it has to check few URLs is acceptable compared to the rest of the versions. As we increase the number of URLs the total time increases exponentially. We also observe a very high number of calls, where the vast majority are operations to manage the request. If we look at the *asynchronous strategy*, we can see how its time remains quite low and stable, as well as the number of calls required. On the other hand, for the *version with threads*, the times are not quite good, it seems that a low number of URLs per message does not get a good performance, since from 8 URLs per message is where we start to see the difference.

So, why is the threaded version getting worst time than the asynchronous version even if the number of cores and threads are far enough? A posible explanation would be the **G**lobal **I**nterpreter **L**ock (GIL)[14]. The GIL prevents two threads from executing simultaneously in the same program, even in a multi-threaded architecture with more than one **C**entral **P**rocessing **U**nit (CPU) core.

As a conclusion, we will stay with the **asynchronous strategy**, since in terms of time it has the best results and its level of calls is also quite good, compared to the initial synchronous version. The version with threads starts to be useful when there are many URLs inside the message and we have to check them all, otherwise it seems to be penalized at time level and adds a load at thread level in our system.

Once at this point we could think about using some optimization libraries like Cython or Numba, but if we take a look at the results provided by the profiler we see that it does not make much sense to add variable typing to improve the speedup, since as previously mentioned, the main bottleneck of the system are HTTP requests and it is not worth adding module compilation and new syntax in the code if we can not get a significant increase in the speedup of the system. We will leave these improvements for when the system grows and new needs arise, where it makes sense to apply these improvements.

[14]The GIL, in simple words, is a mutex (or a lock) that allows only one thread to hold the control of the Python interpreter.

5. Deployment

In this chapter, we will explain how the system deployment phase has been performed on a remote server. When we talk about system deployment we mean taking a local application, in this case in Python, and providing enough means to the client to be able to use that application as an API resource or web page. As we have already explained in previous chapters, our goal is to provide an api to validate if the content of an SMS is legitimate or illegitimate.

Following this idea, we must start by opening an **S**ecure **SH**ell (SSH) connection to the remote server. After that, we will do an analysis of the server characteristics where we are going to deploy our application to ensure that the machine does not lack resources and guarantee a correct installation of the working modules.

Once this analysis is done, we will go on configuring the environment in which the system will work, this will involve all the installation of libraries and modules that our Python application needs, from the installation of the tool to create virtual environments to the configuration of the system paths of the new server used to locate specific files, like logging ones.

At this point, we will relaunch all the tests that we elaborated to verify that everything continues working correctly and we have not left anything pending.

If everything is OK, we will go into more detail and we will be able to deploy the system based on the following points:

- **Core**
- **Flask**
- **GUnicorn**
- **Nginx**

The deployment schema is shown in Figure 5.1. In the following sections, each of the abovementioned parts will be explained in more detail.

5.1. Server features

This section will show the main features of the server where we will deploy the Python application. These features will be crucial to efficiently configure each component to take full advantage of our resources.

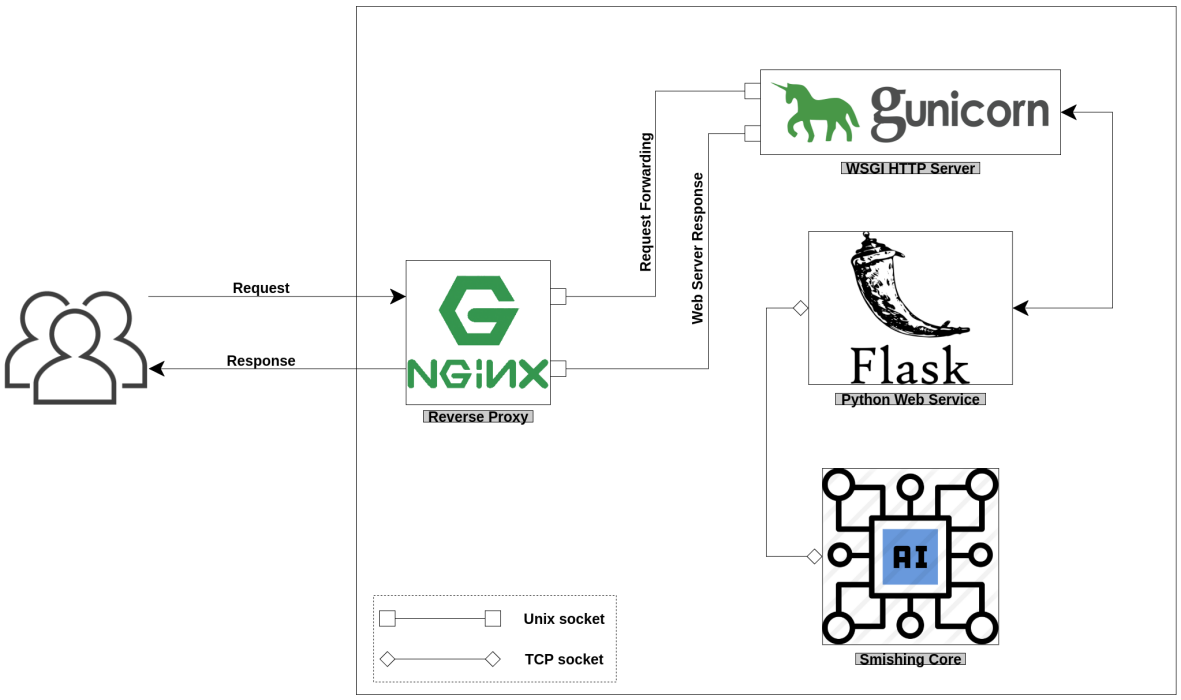

Figure 5.1.: System deployment schema

5.1.1. CPU architecture

To obtain the information about the CPU architecture, we'll use the ***lscpu*** command, in order to gather information from **sysfs** and **/proc/cpuinfo** in an easy readable format by humans. The information includes, for example, the architecture, the number of CPUs, threads, sockets, etc (See Figure 5.2).

5.1.2. System memory

To obtain the information about the physical (**R**andom **A**ccess **M**emory (RAM)) and swap memory, we'll use the **free** command with the **-m** arguments, to indicate that we want the result in Megabytes. It is important to know the resources that the system has, because if they fall short, it will be necessary to increase them or rethink the deployment in another server. As shown in Figure 5.3, with 4GiB of RAM and 1 GiB of swap we have more than enough for our system.

5.1.3. Operating system

Another important aspect to know is the type of operating system to be used and its version, since the installation of some packages and libraries may vary depending on it. Since we already know a priori that the system is a **CentOS**, we will use the ***cat*** command to display the information in the **/etc/centos-release** file, resulting as follows:

```
CentOS Linux release 8.3.2011
```

```
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              2
On-line CPU(s) list: 0,1
Thread(s) per core:  1
Core(s) per socket:  1
Socket(s):           2
NUMA node(s):        1
Vendor ID:           GenuineIntel
CPU family:          6
Model:               63
Model name:          Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz
Stepping:            0
CPU MHz:             2394.375
BogoMIPS:            4788.75
Hypervisor vendor:   VMware
Virtualization type: full
L1d cache:           32K
L1i cache:           32K
L2 cache:            1024K
L3 cache:            28160K
NUMA node0 CPU(s):   0,1
Flags:               fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx
 fxsr sse sse2 ss syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon nopl xtopology tsc_reliable non
stop_tsc cpuid pni pclmulqdq ssse3 fma cx16 pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer ae
s xsave avx f16c rdrand hypervisor lahf_lm abm cpuid_fault invpcid_single pti ssbd ibrs ibpb stibp fsgsb
ase tsc_adjust bmi1 avx2 smep bmi2 invpcid xsaveopt arat md_clear flush_l1d arch_capabilities
```

Figure 5.2.: Server CPU architecture

```
              total        used        free      shared  buff/cache   available
Mem:           3903        1001        1634          27        1268        2641
Swap:          1023          13        1010
```

Figure 5.3.: Server physical and swap memory

5.2. Environment preparation

In this section we will explain the steps followed for the initial preparation of the environment where the system will have to work. We will cover from the version checking of the existing tools in the server to the installation and testing of the modules that will conform the system reusing the previously elaborated tests.

5.2.1. Server tools versions

We will mainly focus on ensuring that the important tools to work with are installed and in the desired version. The versions obtained are as follows:

- **python**: Python 3.8.3
- **pip**: pip 21.1.1
- **git**: git 2.27.0

Fortunately, the server already had the minimum requirements to start cloning the repository.

5.2.2. Git repository clone

We will clone the GIT repository in the path **/opt/git** of the server using the command **git clone** with the https option and check that it has been done correctly doing a **ls** inside the project directory.

```
[rtruchero@lleidanet.lnst.es@devel3 smishing]$ ll
total 12
drwxr-xr-x.  4 rtruchero@lleidanet.lnst.es apps-core@lleidanet.lnst.es   26 may 19 13:45 datasets
drwxr-xr-x.  2 rtruchero@lleidanet.lnst.es apps-core@lleidanet.lnst.es 4096 may 19 13:45 notebooks
-rw-r--r--.  1 rtruchero@lleidanet.lnst.es apps-core@lleidanet.lnst.es  700 may 19 13:45 README.md
-rw-r--r--.  1 rtruchero@lleidanet.lnst.es apps-core@lleidanet.lnst.es  757 may 19 18:32 requirements.txt
drwxr-xr-x. 10 rtruchero@lleidanet.lnst.es apps-core@lleidanet.lnst.es  139 may 27 11:50 system
drwxr-xr-x.  5 rtruchero@lleidanet.lnst.es apps-core@lleidanet.lnst.es   53 may 19 13:45 theory
```

Figure 5.4.: Git project extraction checking

For better navigability and readability when using the project paths, a symbolic link is made to the main project folder, as follows:

```
[rtruchero@lleidanet.lnst.es@devel3 ~]$ sudo ln -s /opt/git/SmishingDetector /opt/smishing
```

5.2.3. Virtual environment configuration

To configure the virtual environment, we must first install the **virtualenv** utility on the system using **P**ip **I**nstalls **P**ackages (PIP), the Python package installer. This virtual environment tool creates a folder inside the project directory. By default, the folder is named **venv**, but it is possible to rename it as well. When the virtual environment is activated, the packages installed after that are installed inside the virtual environment folder of the specific

project.

So, we installed **virtualenv**, checked it's version, created an specific Python3 virtualenv and activated to go to the next step:

```
[rtruchero@lleidanet.lnst.es@devel3 smishing]$ pip3 install virtualenv
[rtruchero@lleidanet.lnst.es@devel3 smishing]$ virtualenv --version
virtualenv 20.4.7 from /home/rtruchero@lleidanet.lnst.es/.local/lib/python3.8/↩
    ↪ site-packages/virtualenv/__init__.py
[rtruchero@lleidanet.lnst.es@devel3 smishing]$ virtualenv -p python3.8 venv
[rtruchero@lleidanet.lnst.es@devel3 smishing]$ source venv/bin/activate
(venv) [rtruchero@lleidanet.lnst.es@devel3 SmishingDetector]$
```

5.2.4. Server paths configuration

Since some paths have changed with respect to the local version, inside the constants file we will redefine the variable **MAIN_PATH** with the current path of the classes in the server. To keep the import structure we will also add the following line inside the configuration file of the virtual environment we are going to work under (**venv/bin/activate**):

```
export PYTHONPATH="/opt/smishing/system"
```

With this simple line. we will be able to define the root directory in order to set up a consistent and orderly structure for importing classes and modules.

5.2.5. Pip libraries installation

To avoid having to install each python library one by one, we will use the **-r requirements_file** option provided by the **pip** command. This way we will install all the python libraries in one simple step.

The installed python libraries are:

```
aiohttp==3.7.4.post0
async-timeout==3.0.1
attrs==21.2.0
beautifulsoup4==4.9.3
bs4==0.0.1
certifi==2020.12.5
chardet==4.0.0
click==8.0.0
filelock==3.0.12
Flask==2.0.0
future==0.18.2
gunicorn==20.1.0
idna==2.10
iniconfig==1.1.1
itsdangerous==2.0.1
Jinja2==3.0.1
```

```
joblib==1.0.1
MarkupSafe==2.0.1
Morfessor==2.0.6
multidict==5.1.0
mysqlclient==2.0.3
nltk==3.6.2
numpy==1.20.3
packaging==20.9
pluggy==0.13.1
polyglot==16.7.4
py==1.10.0
pycld2==0.41
PyICU==2.7.3
pyparsing==2.4.7
pytest==6.2.4
python-whois==0.7.3
regex==2021.4.4
requests==2.25.1
requests-file==1.5.1
scikit-learn==0.24.2
scipy==1.6.3
six==1.16.0
sklearn==0.0
soupsieve==2.2.1
threadpoolctl==2.1.0
tldextract==3.1.0
toml==0.10.2
tqdm==4.60.0
typing-extensions==3.10.0.0
urllib3==1.26.4
Werkzeug==2.0.1
yarl==1.6.3
```

It should be noted that the installation has been done with the virtual environment active, so we do not create dependencies with other projects that may exist on the server.

5.2.6. Database schema deployment

In a similar way as we have done in the 5.2.4 section, we will have to redefine the variables inside the constants file that refer to the HOST, PORT, USER, PASSWORD and SCHEMA of the MYSQL database, so that the system can perform the corresponding operations. It should be added that prior to this, the systems team has deployed a MYSQL service within the server in order to have an operational relational database.

Next we will relaunch the script in Annex B to create the new database schema and its corresponding tables, and check all has been created without any issues.

5.2.7. Test validations

Once all the preparations have been made for the correct deployment, we re-launch all the unit, functional and integration tests implemented, shown in subsection 4.3.1, with a single execution pipeline. In this way we will notice if any component fails and in subsequent deployment phases we will not introduce errors caused by other issues.

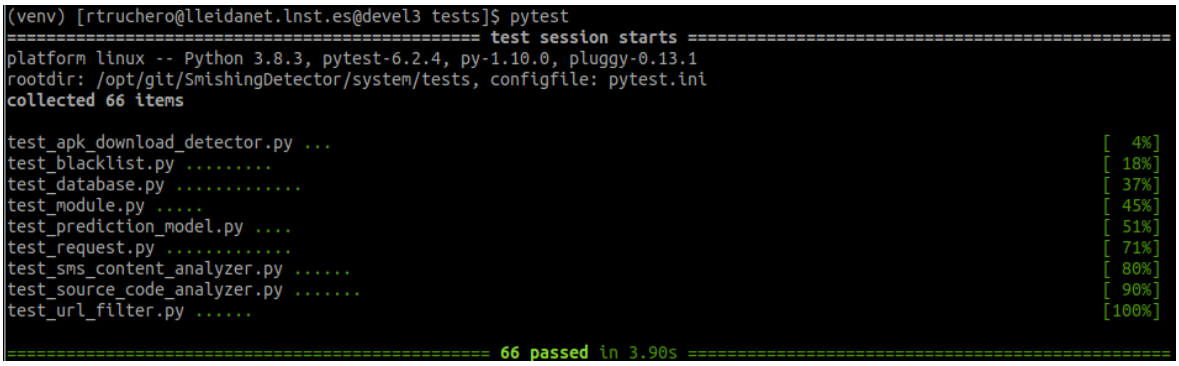

Figure 5.5.: Single pytest pipeline output

5.3. Core

To start with the deployment and make life easier in future stages, we will launch the Engine process as a service that will be listening for requests through a TCP socket so that it works independently from the API. To do this we will start by creating 3 new files:

1. **smishing-core.service**: Core service configuration file. (See Annex C.1.1)

2. **smishing.sh**: Simple shell that will try to launch indefinitely every 10 seconds the smishing.py file, and will redirect all the error outputs to a debug file. (See Annex C.1.2)

3. **smishing.py**: Main Python script that try to start the Engine and keep waiting requests. This scripts will also manage all the signals received and finish tasks safely, e.g when we stop the service the system will send a SIGTERM signal to kill the process, our script will catch it and stop the Core safely. (See Annex C.1.3)

Next, we'll start the service file that we created and we will check it by launching with no errors. If does, enable it, to automatically start the service if the server reboots or goes down.

```
[rtruchero@lleidanet.lnst.es@devel3 ~]$ start_core
smishing-core.service - Smishing Core
  Loaded: loaded (/etc/systemd/system/smishing-core.service; enabled; vendor ↩
      ↪ preset: disabled)
  Active: active (running) since Tue 2021-05-27 12:20:23 CEST;
Main PID: 1541904 (smishing.sh)
   Tasks: 6 (limit: 24735)
```

```
  Memory: 79.5M
  CGroup: /system.slice/smishing-core.service
          1541904 /bin/bash /opt/smishing/system/smishing.sh
          1541905 /bin/bash /opt/smishing/system/smishing.sh
          1541906 python ./smishing.py
          1541907 /bin/tee -a /opt/localstore/logs/smishing.debug

may 27 12:20:23 devel3.lnst.es systemd[1]: Started Smishing Core.
```

In order to check if the core is up an running, we can review the process ID and the logs.

```
[rtruchero@lleidanet.lnst.es@devel3 ~]$ ps afx | grep smishing
1543736 pts/0 S+ 0:00 \_ grep --color=auto smishing
1541904 ? Ss 0:00 /bin/bash /opt/smishing/system/smishing.sh
1541905 ? S 0:00 \_ /bin/bash /opt/smishing/system/smishing.sh
1541906 ? Sl 0:01 \_ python ./smishing.py
1541907 ? S 0:00 \_ /bin/tee -a /opt/localstore/logs/smishing.debug

...

2021-27-05 12:20:24,865,865 INFO 841f3e9f623c4402a29ba9b0e933b52f smishing.py↩
    ↪ :45 time:1622542824 msg: STARTING CORE!
```

5.4. Flask

5.4.1. What is?

Flask is a minimalistic framework written in Python that allows you to create web applications quickly and with a minimum number of lines of code. It is based on the Werkzeug **W**eb **S**erver **G**ateway **I**nterface (WSGI) specification and the Jinja2 template engine and has a **B**erkeley **S**oftware **D**istribution (BSD) license.

For our personal case we will use it to build the skeleton of calls to the API, **HTTP POST validate** and **HTTP GET results** methods.

5.4.2. Installation

In order to be able to use the functionalities that Flask offers us, first we will have to install it inside our virtual environment in which we will run the system. In our case we already have it installed in subsection 5.2.5, but if we would want it to install we should use the following pip command:

```
(venv) [rtruchero@lleidanet.lnst.es@devel3 ~]$ pip install flask
Collecting flask
  Downloading Flask-2.0.1-py3-none-any.whl (94 kB)
     || 94 kB 1.2 MB/s
Collecting Jinja2>=3.0
  Using cached Jinja2-3.0.1-py3-none-any.whl (133 kB)
```

```
Collecting itsdangerous>=2.0
  Using cached itsdangerous-2.0.1-py3-none-any.whl (18 kB)
Collecting Werkzeug>=2.0
  Using cached Werkzeug-2.0.1-py3-none-any.whl (288 kB)
Collecting click>=7.1.2
  Downloading click-8.0.1-py3-none-any.whl (97 kB)
     || 97 kB 4.4 MB/s
Collecting MarkupSafe>=2.0
  Using cached MarkupSafe-2.0.1-cp38-cp38-manylinux2010_x86_64.whl (30 kB)
Installing collected packages: MarkupSafe, Werkzeug, Jinja2, itsdangerous, click, flask
Successfully installed Jinja2-3.0.1 MarkupSafe-2.0.1 Werkzeug-2.0.1 click-8.0.1 flask-2.0.1 itsdangerous-2.0.1
```

In addition to this package, if we do not have them installed, other packages necessary for Flask to work correctly will also be installed.

5.4.3. Configuration

Because of the testing needs it has already been implemented previously and the architecture has been explained in subsection 4.2.2.

5.4.4. Testing

As this will be the first layer of our deployment, we will test that everything is working fine. For this we deploy the Flask server and launch a couple of **c**lient **URL** (CURL) requests to the API methods.

```
(venv) [rtruchero@lleidanet.lnst.es@devel3 api]$ python Request.py
 * Serving Flask app 'Request' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [28/May/2021 10:47:18] "POST /validate HTTP/1.1" 200 -
127.0.0.1 - - [28/May/2021 10:48:05] "GET /results/?id=162211832944433 HTTP/1.1" 200 -
```

Figure 5.6.: Flask application server running

```
(venv) [rtruchero@lleidanet.lnst.es@devel3 api]$ curl --location --request POST 'http://127.0.0.1:5000/validate' \
> --header 'Accept: application/json' \
> --header 'Content-Type: application/json' \
> --data-raw '{
>     "id_mt" : 161616161,
>     "id_user" : 123456,
>     "text" : "449050000301 You have won a £2,000 price! To claim, call 09050000301."
> }'
{"code":"200","id":162219163886711,"status":"Success"}
```

Figure 5.7.: Flask server curl test with API POST validate method

```
(venv) [rtruchero@lleidanet.lnst.es@devel3 api]$ curl --location --request GET 'http://127.0.0.1:5000/results/?id=162211832944433' \
> --header 'Accept: application/json' \
> --header 'Content-Type: application/json'
{"code":"200","id":162211832944433,"id_mt":161616161,"legitimate":0,"request_status_code":4,"status":"Success"}
```

Figure 5.8.: Flask server curl test with API GET results method

Once everything is checked and working as expected, we can move on to the next phase of the deployment, which will be to add GUnicorn into the equation. This will serve as a bridge between our Python Flask application and the NGINX web server.

5.5. GUnicorn

5.5.1. What is?

Gunicorn is an HTTP server for Unix systems that complies with the WSGI specification. It allows us to serve our Flask application with multiple workers to increase the performance of our application and translate all WSGI responses from the Python Flask application into HTTP responses suitable for the NGINX server to understand. Without GUnicorn, the communication between these two components would not be possible.

5.5.2. Installation

Similar to what we did with the Flask framework, we will install the Gunicorn tool through the PIP using the following command:

```
(venv) [rtruchero@lleidanet.lnst.es@devel3 ~]$ pip install gunicorn
Collecting gunicorn
  Using cached gunicorn-20.1.0-py3-none-any.whl (79 kB)
Requirement already satisfied: setuptools>=3.0 in ./test/lib/python3.8/site-packages (from gunicorn) (56.0.0)
Installing collected packages: gunicorn
Successfully installed gunicorn-20.1.0
(test) [rtruchero@lleidanet.lnst.es@devel
```

5.5.3. Configuration

As we already know, Gunicorn is a Python WSGI HTTP Server that supports the WSGI protocol. To start using its advantages and prepare for the third phase (NGINX), we will have to create a Python file, e.g. **wsgi.py** that will instantiate our Flask application and executes it:

```
from api.Request import app

if __name__ == "__main__":
    app.run()
```

At this point, we could just run gunicorn executing:

```
(venv) [rtruchero@lleidanet.lnst.es@devel3 api]$ gunicorn --bind 0.0.0.0:5000 wsgi:app
```

and we will have a functional gunicorn server running, but this is not the best practice since a reboot of the server will take down our application. For that reason we are interested on running the gunicorn instance as a systemd service. To do that, we'll create the service file **/etc/systemd/system/smishing-api.service** (See C.2) and save it. The next step is to init the service and if does correctly enable it, in order to start the service if the server reboots or goes down.

```
[rtruchero@lleidanet.lnst.es@devel3 ~]$ sudo systemctl start smishing-api && sudo systemctl status smishing-api
 smishing-api.service - Gunicorn instance to serve smishing project
   Loaded: loaded (/etc/systemd/system/smishing-api.service; disabled; vendor preset: disabled)
   Active: active (running) since Fri 2021-05-28 13:33:31 CEST;
 Main PID: 1070220 (gunicorn)
    Tasks: 1 (limit: 24735)
   Memory: 5.1M
   CGroup: /system.slice/smishing-api.service
           1070220 /opt/git/SmishingDetector/system/venv/bin/python /opt/smishing/system/venv/bin/gunicorn --wo>

may 28 13:33:31 devel3.lnst.es systemd[1]: Started Gunicorn instance to serve smishing project.

[rtruchero@lleidanet.lnst.es@devel3 ~]$ sudo systemctl enable smishing-api
```

5.5.4. Testing

In a similar way as we did with the Flask application, we will launch a couple of requests to the desired **I**nternet **P**rotocol (IP), in this case the 0.0.0.0 on port 5000. Before that we will have to run a separate Gunicorn process, since our service **smishing-api** establishes a communication by socket, that is to say the requests to the Gunicorn server will not go by HTTP but by an internal socket of the system located in **/var/run/smishing.sock**, this reinforces enormously the speed and security of the communications.

```
(venv) [rtruchero@lleidanet.lnst.es@devel3 api]$ gunicorn --bind 0.0.0.0:5000 api.wsgi:app
[2021-05-28 13:56:52 +0200] [1070480] [INFO] Starting gunicorn 20.1.0
[2021-05-28 13:56:52 +0200] [1070480] [INFO] Listening at: http://0.0.0.0:5000 (1070480)
[2021-05-28 13:56:52 +0200] [1070480] [INFO] Using worker: sync
[2021-05-28 13:56:52 +0200] [1070482] [INFO] Booting worker with pid: 1070482
```

Figure 5.9.: Gunicorn application server running

At this point we're ready to move to the next step and configure our NGINX web server.

```
(venv) [rtruchero@lleidanet.lnst.es@devel3 ~]$ curl --location --request POST 'http://0.0.0.0:5000/validate' \
> --header 'Accept: application/json' \
> --header 'Content-Type: application/json' \
> --data-raw '{
>     "id_mt" : 262626262,
>     "id_user" : 123456,
>     "text" : "449050000301 You have won a £2,000 price! To claim, call 09050000301."
> }'
{"code":"200","id":162220302488185,"status":"Success"}
```

Figure 5.10.: Gunicorn server curl test with API POST validate method

```
(venv) [rtruchero@lleidanet.lnst.es@devel3 ~]$ curl --location --request GET 'http://0.0.0.0:5000/results/?id=162220302488185' \
> --header 'Accept: application/json' \
> --header 'Content-Type: application/json'
{"code":"200","id":162220302488185,"id_mt":262626262,"legitimate":0,"request_status_code":4,"status":"Success"}
```

Figure 5.11.: Gunicorn server curl test with API GET results method

5.6. NGINX

5.6.1. What is?

NGINX, pronounced ”engine-ex”, is a popular open source web server software. In its initial version, it worked on HTTP web servers. However, today it also serves as a reverse proxy, HTTP load balancer and email proxy for **I**nternet **M**essage **A**ccess **P**rotocol (IMAP), **P**ost **O**ffice **P**rotocol (POP3) and **S**imple **M**ail **T**ransfer **P**rotocol (SMTP).

The main objective will be to use NGINX only as a reverse proxy, since we only have an application server and the rest of the options do not interest us.

The structure of the software is asynchronous and event-driven, which allows the processing of many requests at the same time. NGINX is also highly scalable, which means that its services grow with the traffic of its clients, making it one of the best web servers on the market.

5.6.2. Installation

To start configuring and using NGINX together with our Gunicorn and Flask components, we must install it on the server and check its version. In this case using the following command:

```
(venv) [rtruchero@lleidanet.lnst.es@devel3 smishing]$ sudo dnf install nginx
Última comprobación de caducidad de metadatos hecha hace 1:11:11, el jue 27 ↩
    ↪ may 2021 11:12:53 CEST.
El paquete nginx-1:1.14.1-9.module_el8.0.0+184+e34fea82.x86_64 ya está ↩
    ↪ instalado.
Dependencias resueltas.
Nada por hacer.
¡Listo!
(venv) [rtruchero@lleidanet.lnst.es@devel3 system]$ nginx -v
nginx version: nginx/1.14.1
```

As we can see, the NGINX version installed by default in our CentOS 8 is 1.14.1. This is

not the latest stable version and therefore, there are some vulnerabilities on it [1]. That is why we first need to create in the folder **/etc/yum.repos.d/** the file **nginx.repo** with the configurations to be able to download the latest stable version of NGINX:

```
[nginx-stable]
name=nginx stable repo
baseurl=http://nginx.org/packages/centos/$releasever/$basearch/
gpgcheck=1
enabled=1
gpgkey=https://nginx.org/keys/nginx_signing.key
module_hotfixes=true

[nginx-mainline]
name=nginx mainline repo
baseurl=http://nginx.org/packages/mainline/centos/$releasever/$basearch/
gpgcheck=1
enabled=0
gpgkey=https://nginx.org/keys/nginx_signing.key
module_hotfixes=true
```

Once this is done, we will redo the NGINX installation and check if the latest stable version has been installed.

```
(venv) [rtruchero@lleidanet.lnst.es@devel3 system]$ sudo dnf install nginx
...
Actualizado:
  nginx-1:1.20.1-1.el8.ngx.x86_64

¡Listo!
(venv) [rtruchero@lleidanet.lnst.es@devel3 system]$ nginx -v
nginx version: nginx/1.20.1
```

For the moment no vulnerabilities have been detected for NGINX version 1.20.1[2], so we can continue with the configuration.

5.6.3. Configuration

Once we have all the necessary tools installed, it is time to configure our web server to suit our needs. The options used in the Annex C.3.2 configuration file, which will be explained in more detail below.

```
user root;
```

Defines the user and group credentials used by worker processes. If group is omitted, a group whose name equals that of user is used.

[1] https://nvd.nist.gov/vuln/search/results?form_type=Advanced&cves=on&cpe_version=cpe:/a:nginx:nginx:1.14.0#:~:text=CVE-2018-16844-,nginx%20before%20versions%201.15.,used%20in%20a%20configuration%20file.

[2] https://nvd.nist.gov/vuln/search/results?form_type=Advanced&results_type=overview&query=nginx+1.20.1&search_type=all

```
worker_processes 2;
```

The best number of workers to configure is equal to the number of core's of the system. So if we look at Figure 5.2 we can see that this value is 2.

```
error_log /opt/localstore/logs/error.log;
```

Reconfigure the error.log location. If NGINX faces any glitches then it will record the event to the error log. This may happen if there are some errors in the configuration file. Therefore, if NGINX is unable to start or abruptly stopped running then we should check the error logs to find more details. A few warning may also will be found in the error log but it does not indicate that a problem has occurred but the event may pose a serious issue in the near future.

```
pid /var/run/nginx.pid;
```

Reconfigure the **P**rocess **ID**entificator (PID) location. It is cleaner to place the pids inside the /var/run directory.

```
events {
   worker_connections 1024;
}
```

Sets the maximum number of simultaneous connections that can be opened by a worker process. It should be kept in mind that this number includes all connections (e.g. connections with proxy servers, among others), not only connections with clients. Another consideration is that the actual number of simultaneous connections cannot exceed the current limit on the maximum number of open files, which can be changed by **worker__rlimit__nofile**. In order to check it we'll run the following command:

```
[rtruchero@lleidanet.lnst.es@devel3 logs]$ ulimit -n
1024
```

So the best value is 1024 worker connections.

All the previous instructions were related to the root NGINX configs, if we want an http web server we need to configure an http block with more instructions within.

```
http {
   #...
}
```

So, hereafter all the instructions will be within this http context block.

```
   include mime.types;
   include proxy.conf;
```

The include statement allows us to import data from a file into our NGINX configuration. This is especially important if you want to have a structured and readable configuration file. The first include simply imports all the basic MIME types while the second contains a basic configuration of some important proxy parameters.

```

    log_format main '$remote_addr - $remote_user [$time_local] $status '
        '"$request" $body_bytes_sent "$http_referer" '
        '"$http_user_agent" "$http_x_forwarded_for"';

    access_log /opt/localstore/logs/access.log main;
```

The first instruction just defines a log format, we can customize it as we want. The second instruction redefines the **access.log location** and specifies that all logs in there will follow the previous *main* log format.

```
server_tokens off;
```

Hides the NGINX server version on responses. Some attackers can use this information to exploit possible server vulnerabilities, so it's very important to disable this parameter.

```
server {
       # Server fundamentals
       listen 80 default_server;
       listen [::]:80 default_server;
       server_name devel3.lnst.es;

       location / {
           return 308 https://$host$request_uri;
       }
}
```

This is the HTTP server configuration block. The main objective of this part is to keep listening at port 80 for HTTP requests and redirect it as **H**yper**T**ext **T**ransfer **P**rotocol **S**ecure (HTTPS) traffic. By doing this, we're assuming that all the requests may be treated as HTTPS and will require HTTPS specifications.

For the HTTPS server block we'll implement a reverse proxy and configure the **S**ecure **S**ockets **L**ayer (SSL) parameters. All the instructions that will be shown below are inside another server block, in this case the one that acts as Reverse Proxy.

```
       listen 443 ssl http2;
       listen [::]:443 ssl http2;
       server_name devel3.lnst.es;
```

To accept the HTTPS traffic, we'll listen SSL on port 443 and define the server name **U**niform **R**esource **I**dentifier (URI). We can define other ports to listen this traffic, but we will use

these since they are the standard.

```
1       ssl_certificate /etc/nginx/ssl/certificate.crt;
2       ssl_certificate_key /etc/nginx/ssl/private.key;
```

The server certificate is a public entity. It is sent to every client that connects to the server. The private key is a secure entity and should be stored in a file with restricted access, however, it must be readable by NGINX's master process. Although the certificate and the key are stored in one file, only the certificate is sent to a client.

```
1       ssl_session_cache shared:MozSSL:10m;
2       ssl_session_timeout 1d;
```

The sessions are stored in an SSL session cache shared between workers and configured by the SSL_session_cache directive. One Megabyte of the cache contains about 4000 sessions. The default cache timeout is 5 minutes. It can be increased by using the **ssl_session_timeout** directive.

```
1       ssl_session_tickets off;
```

It's recommended to disable the SSL session tickets, since we don't want to cache sessions for a long time.

```
1       ssl_protocols TLSv1.2 TLSv1.3;
```

As we well know, SSL is an encryption protocol for the Internet transport layer, its function being to encrypt the data traffic between the client and the server. Since many vulnerabilities have been discovered on it, **T**ransport **L**ayer **S**ecurity (TLS) was born, as an improved version, being more secure, flexible and efficient. That is why we will only use TLS as SSL 2.0 and SSL 3.0 are outdated and not considered secure today.

```
1       ssl_ciphers ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:↩
            ↪ ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-RSA-AES256-GCM-SHA384:ECDHE-↩
            ↪ ECDSA-CHACHA20-POLY1305:ECDHE-RSA-CHACHA20-POLY1305:DHE-RSA-↩
            ↪ AES128-GCM-SHA256:DHE-RSA-AES256-GCM-SHA384;
2       ssl_prefer_server_ciphers on;
```

Defining the SSL protocols and ciphers, we're limiting connections for the ones using strong versions and ciphers of SSL/TLS. So putting the **ssl_prefer_server_ciphers** parameter *on* we are ensuring that when negotatiating which cipher to use with the clients, we prefer using our specified ciphers first.

```
1       ssl_dhparam /etc/nginx/ssl/ffdhe2048.txt;
```

These parameters define how **OpenSSL** performs the **D**iffie-**H**ellman (DH) key-exchange. As we stated correctly they include a field *prime p* and a *generator g*. The purpose of the availability to customize these parameter is to allow everyone to use his/her own parameters for this. This can be used to prevent being affected from the **Logjam attack** [3] (which

[3] **Logjam attack against the TLS protocol**. The Logjam attack allows a man-in-the-middle attacker to

doesn't really apply to 4096 bit field primes) [1].

```
1       add_header X-XSS-Protection "1; mode=block";
```

X-XSS also known as **Cross Site Scripting** header is used to defend against Cross-Site Scripting attacks. XSS Filter is enabled by default in modern web browser such as, Chrome, IE, and Safari. This header stops pages from loading when they detect reflected cross-site scripting (XSS) attacks.

```
1       add_header Strict-Transport-Security "max-age=63072000" always;
```

The **Strict-Transport-Security** header instructs a user agent to only use HTTP connections and it also declared by **Strict-Transport-Security**. This will prevents web browsers from accessing web servers over non-HTTPS connections. Currently all major web browsers support HTTP strict transport security.

```
1       add_header X-Frame-Options DENY;
```

The **X-Frame-Options** header is used to defend websites from clickjacking attack by disabling iframes on your site. Currently it is supported by all major web browser. With this header, you tell the browser not to embed your web page in frame/iframe.

```
1       add_header X-Content-Type-Options nosniff;
```

The **X-Content-Type** header also called *"Browser Sniffing Protection"* to tell the browser to follow the MIME types indicated in the header. It is used to prevent web browser such as, Internet Explorer and Google Chrome from sniffing a response away from the declared Content-Type. nosniff header does not protect all sniffing-related vulnerabilities. Also there is no valid value for this header except **nosniff**.

```
1       location / {
2           limit_except GET POST {
3               deny all;
4           }
5           error_page 403 = @405;
6           proxy_pass http://unix:/var/run/smishing.sock;
7       }
```

First of all, we'll limit the accepted request to GET and POST, since we only have these two valid methods. It's a good practice to limit HTTP methods to the only required. Then convert deny response from **403 (Forbidden)** to **405 (Method Not Allowed)** and finally define the proxy logic to pass all requests to our local unix socket.

downgrade vulnerable TLS connections to 512-bit export-grade cryptography. This allows the attacker to read and modify any data passed over the connection. The attack is reminiscent of the **FREAK attack**, but is due to a flaw in the TLS protocol rather than an implementation vulnerability, and attacks a DH key exchange rather than an **R**ivest **S**hamir & **A**dleman (RSA) key exchange. The attack affects any server that supports **DHE_EXPORT ciphers**, and affects all modern web browsers. 8.4% of the Top 1 Million domains were initially vulnerable.

5.6.4. Testing

At this point we only have to prove that all the pieces of the puzzle work together. For this, we will launch a couple of requests in a similar way to the rest of the previous parts. But in this case on the URL of our deployed API for the HTTP and HTTPS traffic.

5.6.4.1. HTTP tests

We will start with the HTTP traffic only to test that the requests are executed normally, that is, that our NGINX proxy makes the correct redirection of the HTTP traffic to HTTPS.

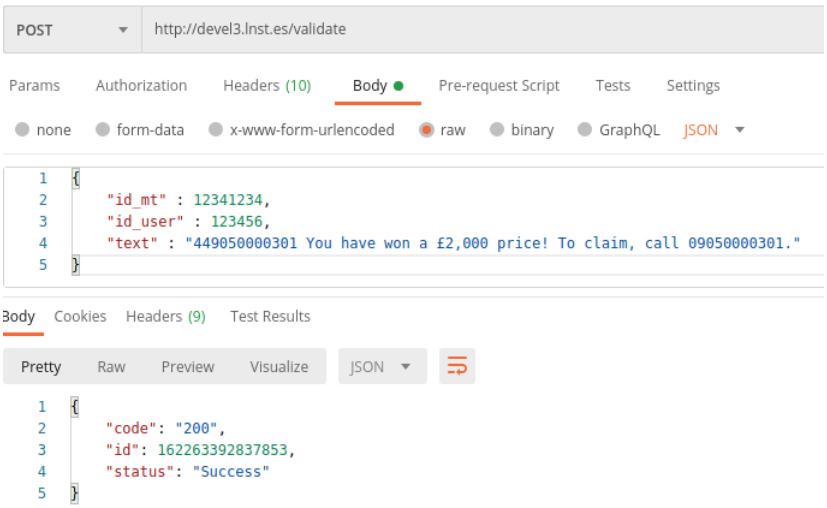

Figure 5.12.: HTTP Nginx validate method

In Figure 5.12, a POST request is made with the HTTP URL and the corresponding JSON body. The response to this request is returned correctly, therefore it means that our NGINX proxy has correctly redirected the request and the corresponding operations have been executed to perform the validation of the SMS.

In Figure 5.13, the GET call is made with the HTTP URL and the identifier obtained in the previous call. As can be seen, the request is processed correctly and the results associated with the same request are returned.

Finally, Figure 5.14 shows the Headers returned by the NGINX server that we have previously configured. We can also check how the version of our web server is not shown, something very important when it comes to protect us against possible attacks.

5.6.4.2. HTTPS tests

In this section, we will test that the HTTPS requests work correctly. The previous tests show that they do, since the only thing we did for the HTTP traffic was to redirect it as HTTPS traffic. Therefore, the requests ended up being executed as if they had been HTTPS from the beginning. It never hurts to test all possible cases to avoid problems in the future.

In Figure 5.15, a POST request is made with the HTTPS URL and the corresponding JSON

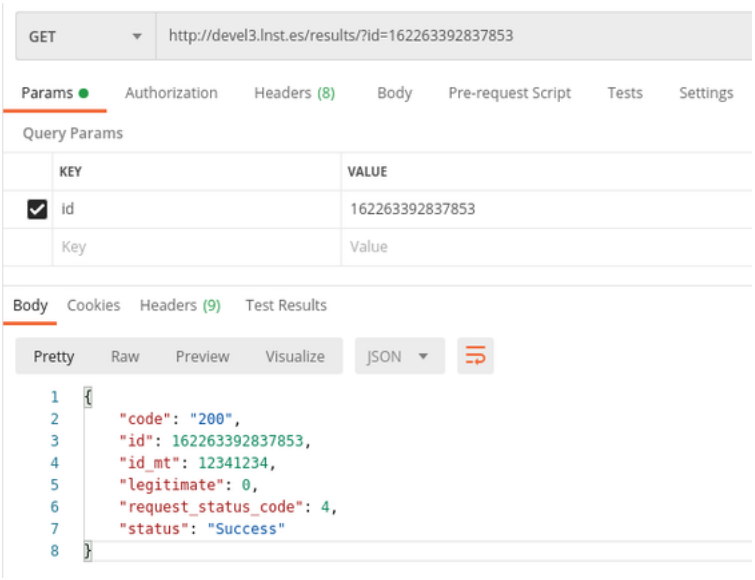

Figure 5.13.: HTTP Nginx results method

KEY	VALUE
Server	nginx
Date	Wed, 02 Jun 2021 11:40:15 GMT
Content-Type	application/json
Content-Length	111
Connection	keep-alive
X-XSS-Protection	1; mode=block
Strict-Transport-Security	max-age=63072000
X-Frame-Options	DENY
X-Content-Type-Options	nosniff

Figure 5.14.: HTTP Nginx response headers

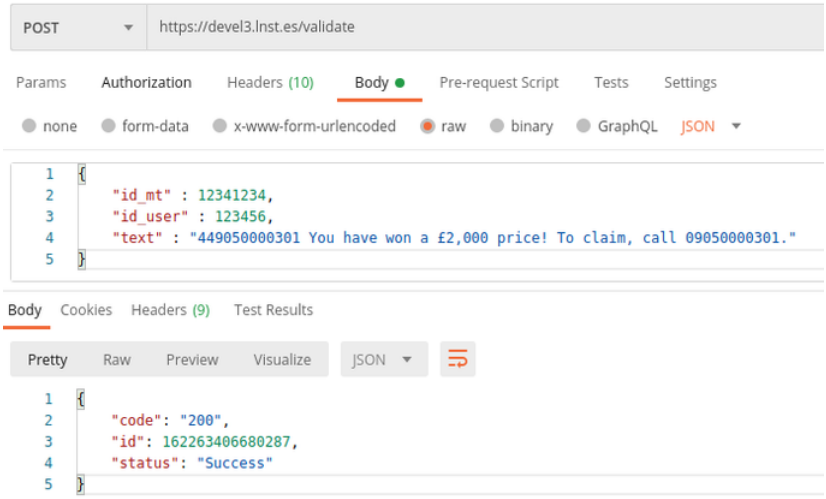

Figure 5.15.: HTTPS Nginx validate method

body. The response to this request is returned correctly and the corresponding operations have been executed to perform the validation of the SMS.

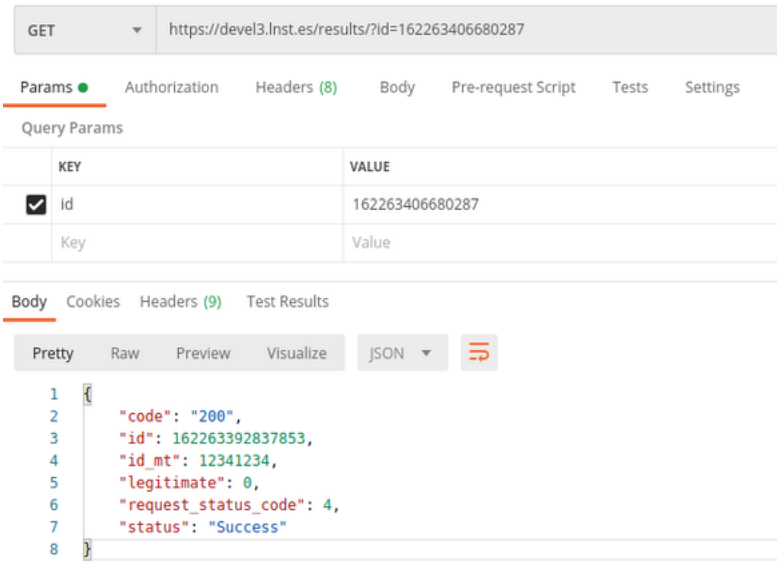

Figure 5.16.: HTTPS Nginx results method

In Figure 5.16, the GET call is made with the HTTPS URL and the identifier obtained in the previous call. As can be seen, the request is processed correctly and the results associated with the same request are returned.

Finally as in the previous subsection, Figure 5.17 shows the Headers returned by the NGINX server that we have previously configured. So, at this point we'll tested the main flows of our application and checked our configuration features.

KEY	VALUE
Server	nginx
Date	Wed, 02 Jun 2021 12:10:12 GMT
Content-Type	application/json
Content-Length	111
Connection	keep-alive
X-XSS-Protection	1; mode=block
Strict-Transport-Security	max-age=63072000
X-Frame-Options	DENY
X-Content-Type-Options	nosniff

Figure 5.17.: HTTPS Nginx response headers

6. Results

This chapter will show the results obtained by performing different tests against the deployed web server API. These tests will range from obtaining ranking metrics based on the degree of success, to measure the response latency of our web server when it is subjected to variable stress, the number of failed connections, and the total processing time of the system's requests.

6.1. Classification metrics

The objective of this section is to evaluate the degree of success of the system, and to extract certain classification metrics that will help us to interpret how our system is working. In section 4.2.3.3, an evaluation similar to the one that will be done in this section has been carried out, but the main difference is that the previous evaluation was only on the prediction model, while this one will be done on the complete functioning of the whole system and its different modules.

To do this, we will create a new script that for each type of message (legitimate and illegitimate) and for each type of language supported (English and Spanish) will launch a finite number of requests against the URL of our service. To lighten the workload, we will divide the tasks into different processes, in our case the following 4:

- **Process 1:** Execute N tests for **English** and **Legitimate** messages.
- **Process 2:** Execute N test for **English** and **Illegitimate** messages.
- **Process 3:** Execute N test for **Spanish** and **Legitimate** messages.
- **Process 4:** Execute N test for **Spanish** and **Illegitimate** messages.

To make life easier, we will add a variable parameter that allows us to configure the number of messages that each process will process. Thus, each of the processes will make all the validation requests and after some time it will retrieve all the results. The objective of all these validations will be to build a **confusion matrix**.

When all the processes have finished their operations, we will put all the results together. We are interested in building on the metrics explained in Annex A for each different language.

In order to carry out the evaluation, 2000 messages have been taken, divided equally according to language and class, i.e. **N=500** messages for each process (4). The results obtained is shown in table 6.1.

At a glance, it is clear that the accuracy has decreased significantly with respect to the evaluation of the prediction model. This is due to the evaluation dataset. For this evaluation

Language	Precision	Recall	F1-Score	Support	Accuracy
EN	0.671683	1	0.803601	1000	0.757085
ES	0.682192	1	0.811075	1000	0.765657

Table 6.1.: System classification metrics

we have used the same amount of legitimate and illegitimate messages, in the evaluation of the different prediction models we could draw a firm conclusion, and that was that our models predicted very well when a message was legitimate, but when a message was illegitimate, in many cases it ended up predicting it as legitimate. Unlike the last evaluation, in this one all the modules of the system come into play, that is why the final accuracy, despite being lower, is not a total disaster.

Another characteristic that we can observe is that for messages in Spanish all the metrics remain the same or slighlty better, even if only by one tenth.

It should also be noted that this evaluation is made with practically empty blacklists, one of the future functionalities will be to add content to the corresponding blacklists when it is detected that it may be illegitimate and to be able to filter these messages better and above all faster.

If we look at the recall metric, we can draw a similar conclusion to the previous evaluation, our system is able to correctly classify and recognize legitimate messages.

6.2. Performance metrics

In this section, we will focus on another type of tests. In the last section, we performed tests to evaluate the degree of accuracy of the system, i.e. how accurately it performed the validations. For the current case, we will be interested in another type of evaluation based on the response time of the requests, in order to finally answer some of the following questions:

- How long do those requests take to complete, including latency for different parts of the request-response cycle?
- How have the requests been going, including how many requests have received non-2xx responses (refers to all requests that weren't correctly received, understood or accepted) and how many have failed?

In order to answer these questions, we could implement another script that independently and concurrently evaluates the different points, but it is not worth reinventing the wheel. In order to achieve our purpose, we have a very useful tool called **Apache Benchmark**. **Apache Benchmark** is a benchmarking tool that measures the performance of a web server by flooding it with HTTP requests and extracting metrics related to **latency** and **success rate**. This tool will allow us to determine how much HTTP traffic our web server can handle before performance begins to decline and establish baselines for typical response times.

This tool allows us to define a set of key parameters that will allow us to simulate the simultaneous connection of clients all of them sending requests until the total number of requests have been sent. We will be interested in testing for the following different cases:

- 1 client and 10 total requests
- 5 clients and 50 total requests
- 10 clients and 100 total requests
- 50 clients and 500 total requests
- 100 clients and 1000 total requests
- 500 clients and 5000 total requests
- 1000 clients and 10000 total requests

All of those tests will be applied for each API method in the system, in our case for:

1. POST /validate
2. GET /results

In addition, we will also get the average response time and the total time of processing all of those requests doing an **S**tructured **Q**uery **L**anguage (SQL) query taking into account the **insert_date** to know when the validation started and the **status_date** to know when the request finished filtering by final states. So, we will not only know the latency of the requests, but the actual time it takes the system to perform the complete processing of that request.

6.2.1. Apache Benchmark POST /validate evaluation

As previously mentioned, in this section all the requests will be made for each different case depending on the number of concurrent clients and total number of requests to be sent. In order to adjust as much as possible to reality and that the requests are executed with the correct sending format, we will create a JSON file where we will specify the body of the message. In this way we will also add its transfer time recreating a real scenario.

The commands used to test each case are the following:

```
ab -c 1 -n 10 -p post_validate.json -r -T application/json https://devel3.lnst.es/validate
ab -c 5 -n 50 -p post_validate.json -r -T application/json https://devel3.lnst.es/validate
ab -c 10 -n 100 -p post_validate.json -r -T application/json https://devel3.lnst.es/validate
ab -c 50 -n 500 -p post_validate.json -r -T application/json https://devel3.lnst.es/validate
ab -c 100 -n 1000 -p post_validate.json -r -T application/json https://devel3.lnst.es/validate
```

```
ab -c 500 -n 5000 -p post_validate.json -r -T application/json https://devel3.lnst.es/validate
ab -c 1000 -n 10000 -p post_validate.json -r -T application/json https://devel3.lnst.es/validate
```

Once the above commands have been executed, all the data returned for each command have been extracted from the system and unified in Table 6.2.

In order to have a more global view of the evolution of the different metrics as we increase the number of concurrent clients and total number of requests, figures 6.1, 6.2 and 6.3 have been generated from table 6.2.

Concurrent Clients	Total number of Requests	GET /results			
		Requests per second [#/sec]	Time per request [ms]	Time per request [ms] (across all concurrent requests)	Failed requests
1	10	61,21	16,338	16,338	0
5	50	172,44	28,995	5,893	0
10	100	180,78	55,316	5,532	0
50	500	83,08	601,807	12,036	0
100	1000	71,19	1404,723	14,047	2
500	5000	609,09	820,894	1,642	4677
1000	10000	739,53	1352,21	1,352	9577

Table 6.2.: Apache Benchmark POST /validate results

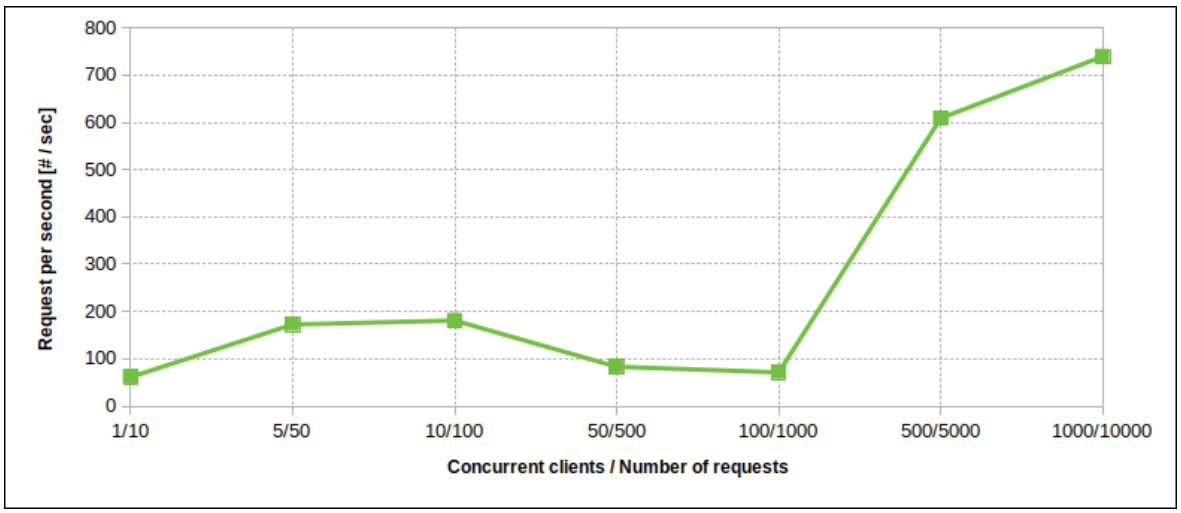

Figure 6.1.: POST /validate requests per second in relation to the concurrent clients and number of requests

In Figure 6.1 we can see how the number of requests per second increases progressively from 61.21 requests per second with 1 client and 10 total requests to 180.78 requests per second with 10 clients and 100 total requests. From this point the number of requests per second decreases to a ratio of 71.19 requests per second with 100 clients and 1000 total requests.

Finally, the progression increases again enormously until reaching a ratio of 739.53 requests per second with 1000 clients and a total of 10000 requests per second.

This huge final variation is due to the number of failed requests where many requests are executed per second but do not necessarily end up in a correct state. We can also see how the optimal ratio of concurrent clients and number of total requests is between the ratios 10/100 and 50/500.

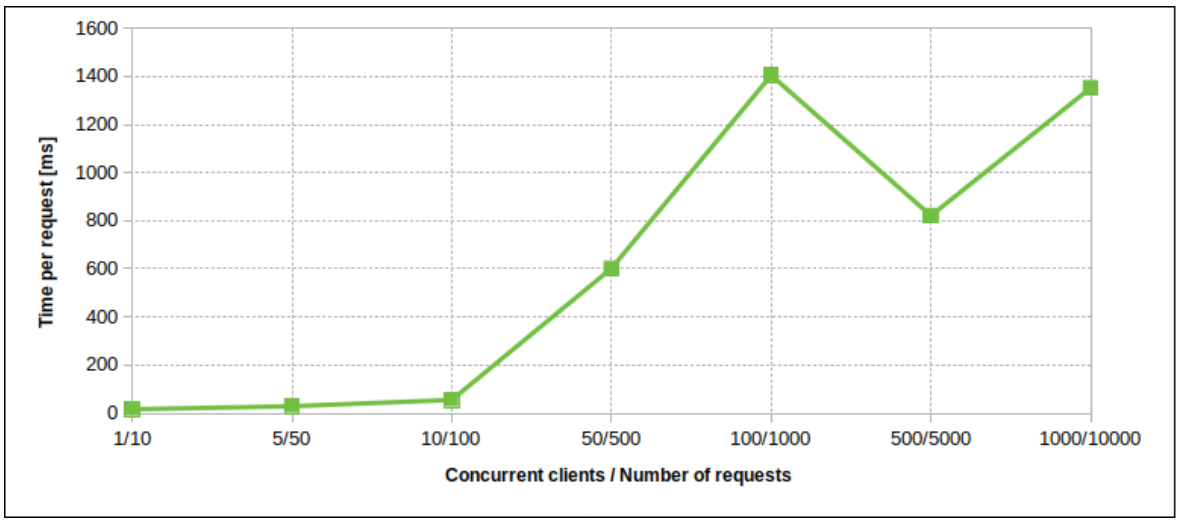

Figure 6.2.: POST /validate time per request in relation to the concurrent clients and number of requests

In Figure 6.2, we can also see that as the proportion of concurrent clients and total number of requests increases, the response time for each request also increases. The variation in the sequence is caused by failed requests. As can be seen in Figure 6.3, after 100 concurrent clients and 1000 requests, the web server starts to mark the requests as failed, this is a consequence of the fact that our NGINX web server cannot process so many requests due to the lack of resources of the machine (See subsection 5.1.1), and therefore a way to accept more simultaneous connections and a higher number of requests would be to increase the server cores and RAM memory.

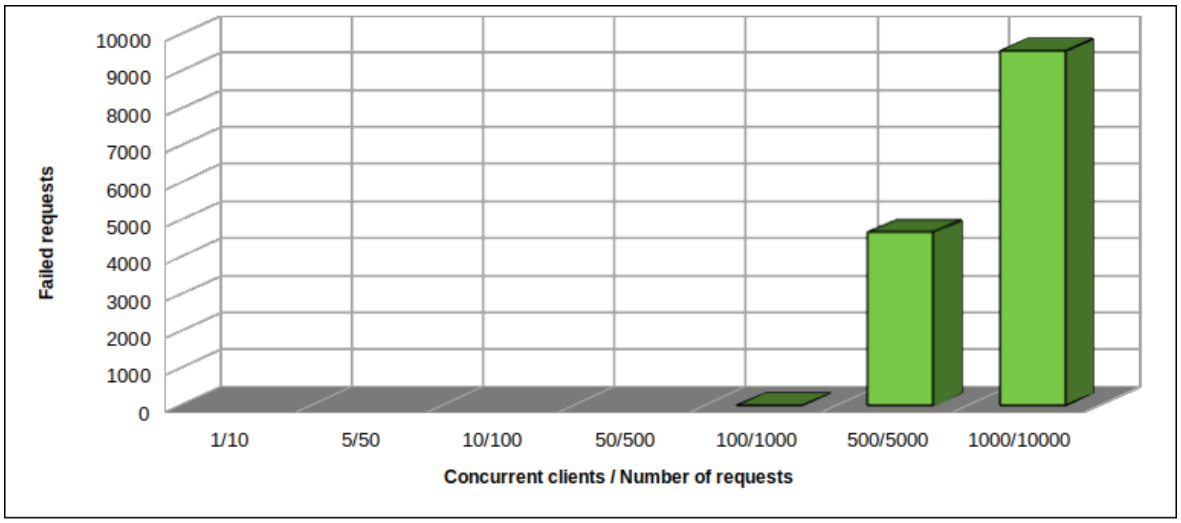

Figure 6.3.: POST /validate failed requests in relation to the concurrent clients and number of requests

6.2.2. Apache Benchmark GET /results evaluation

As in the previous part, we will make requests for each different case depending on the number of concurrent clients and total number of requests to be sent. In this case, we just need to perform an HTTP GET operation to retrieve the results from an specific identifier, e.g. for id **162263392837853**.

The commands used to test each case are the following:

```
ab -c 1 -n 10 -r https://devel3.lnst.es/results/?id=162263392837853
ab -c 5 -n 50 -r https://devel3.lnst.es/results/?id=162263392837853
ab -c 10 -n 100 -r https://devel3.lnst.es/results/?id=162263392837853
ab -c 50 -n 500 -r https://devel3.lnst.es/results/?id=162263392837853
ab -c 100 -n 1000 -r https://devel3.lnst.es/results/?id=162263392837853
ab -c 500 -n 5000 -r https://devel3.lnst.es/results/?id=162263392837853
ab -c 1000 -n 10000 -r https://devel3.lnst.es/results/?id=162263392837853
```

Once the above commands have been executed, all the data returned for each command have been extracted from the system and unified in Table 6.3.

In order to have a more global view of the evolution of the different metrics as we increase the number of concurrent clients and total number of requests, figures 6.4, 6.5 and 6.6 have been generated from table 6.3.

In Figure 6.4 we can already see a notable change in the number of requests per second that our web server can process with respect to Figure 6.1. For 1 client and 10 total requests we obtain a ratio of 271.84 requests per second, which increases quite linearly as we increase the

Concurrent Clients	Total number of Requests	GET /results			
		Requests per second [#/sec]	Time per request [ms]	Time per request [ms] (across all concurrent requests)	Failed requests
1	10	271,84	3,679	3,679	0
5	50	457,59	10,927	2,185	0
10	100	488,53	20,47	2,047	0
50	500	572,94	87,269	1,745	0
100	1000	572,79	174,585	1,746	0
500	5000	687,02	727,785	1,456	1413
1000	10000	711,55	1405,382	1,405	3618

Table 6.3.: Apache Benchmark GET /results results

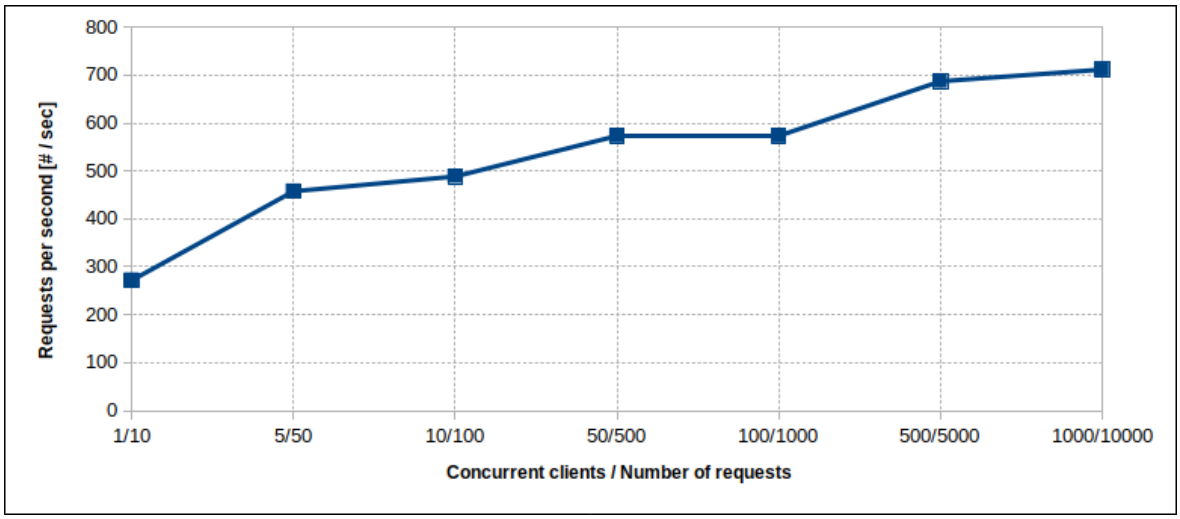

Figure 6.4.: GET /results requests per second in relation to the concurrent clients and number of requests

number of concurrent clients and total requests until reaching a maximum of 711.55 requests per second with 1000 concurrent clients and 10000 total requests.

This is largely due to the fact that the POST /validate method requires a higher internal computation, since it must validate all the fields of the JSON, perform the database insert of the fields and send the request identifier to the Engine to start the validation. While for the GET /results method, only the request identifier format is checked and the validation data is retrieved from the database, whether it is finished or not. Figure 6.5 also shows that as we

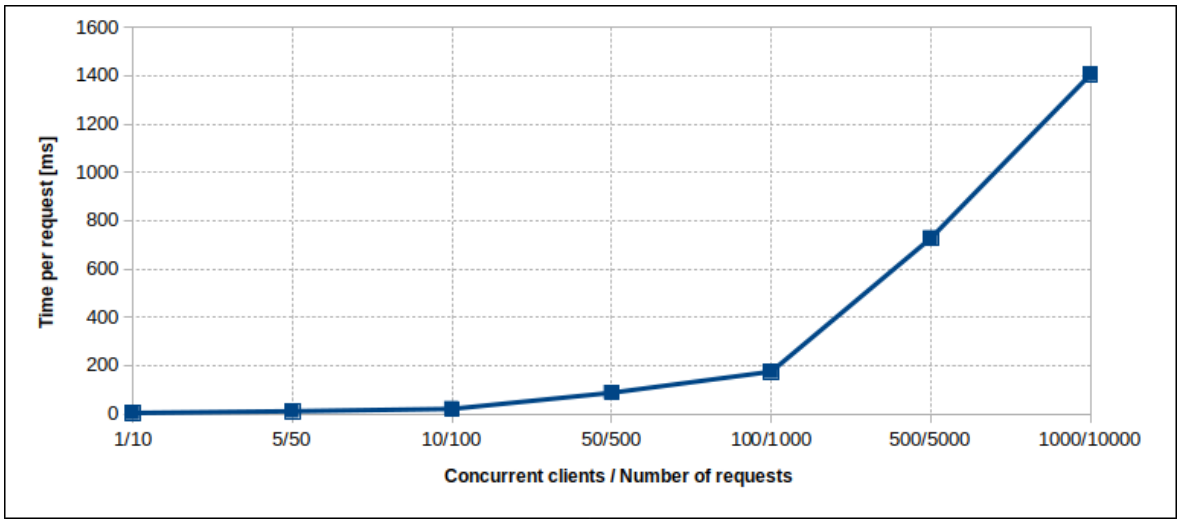

Figure 6.5.: GET /results time per request in relation to the concurrent clients and number of requests

increase the number of concurrent clients and total requests, the response time for each of them also increases. The time increases from 100 concurrent clients and 1000 total requests with 173.585 milliseconds per request until reaching a maximum of 1405.382 milliseconds with 1000 concurrent clients and 10000 total requests.

Finally, in Figure 6.6 we can see how the number of failed requests has decreased significantly with respect to the POST /validate method. As we mentioned before, this is due to the fact that the GET /results method has a lower workload and therefore the server can take on more work as it finishes. Despite this, we see how from 500 concurrent clients and 5000 total requests the failed requests increase up to 1413 reaching a maximum of 3618 with 1000 concurrent clients and 10000 total requests.

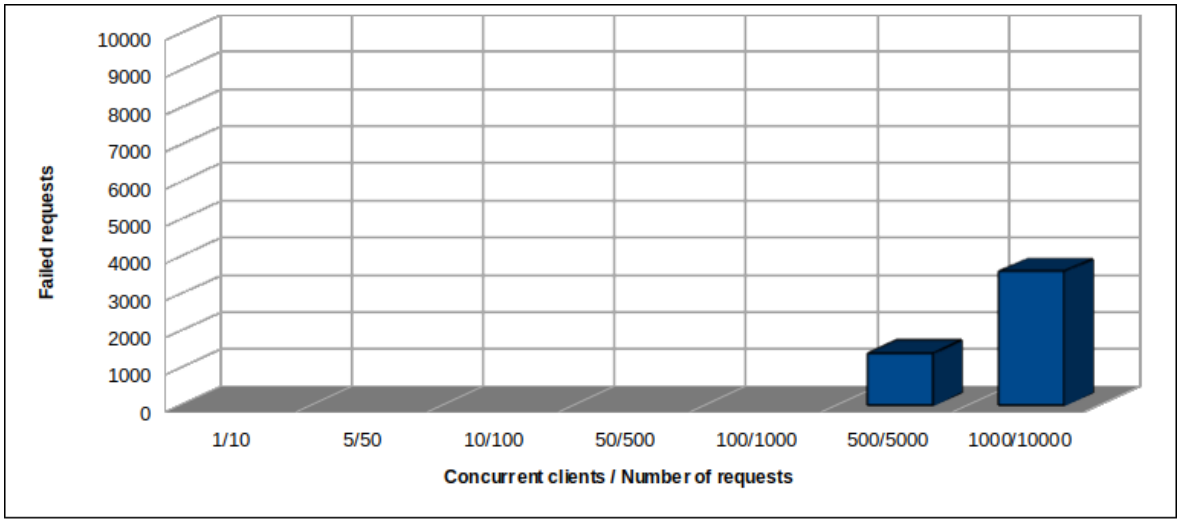

Figure 6.6.: GET /results failed requests in relation to the concurrent clients and number of requests

6.2.3. Validation system evaluation

In this section we will calculate the time that our Core takes for the validation of a request, that is, from the moment in which the first database insert is performed until the database record is updated with the final result of the validation. To do this we will use the following query SQL:

```
SELECT SUM(mean) as average
FROM (
    SELECT status_date-start_date as mean
    FROM smishing.request
    ORDER BY id DESC
    LIMIT N) subq;
```

This query performs a subquery to collect the response times of the last N requests and then calculate the total sum of them. The value of **N** will be the **number of total requests minus the number of failed requests** for that case (for example 10 clients and 100 total requests). In this way we make sure to consider only the response times of the requests that have reached to execute the validations.

In order to cause the desired executions, each execution of the previous subsection 6.2.1 has been used to extract the corresponding data from table 6.4.

As shown in Figure 6.7, as the number of concurrent validations that the Core has to perform increases, the average time for each request grows exponentially, going from 0.2 seconds for 1 client and 10 total requests to approximately 3.38 seconds for 1000 concurrent clients and 10000 total requests, of which only 423 have been executed.

Concurrent Clients	Total number of requests	Successfully pushed requests	System validation time per request [s]	Total system validation time [s]
1	10	10	0,2	2
5	50	50	0,18	9
10	100	100	0,7	70
50	500	500	0,972	486
100	1000	998	1,127	1125
500	5000	323	2,167	700
1000	10000	423	3,383	1431

Table 6.4.: Validation times exctracted from MYSQL database

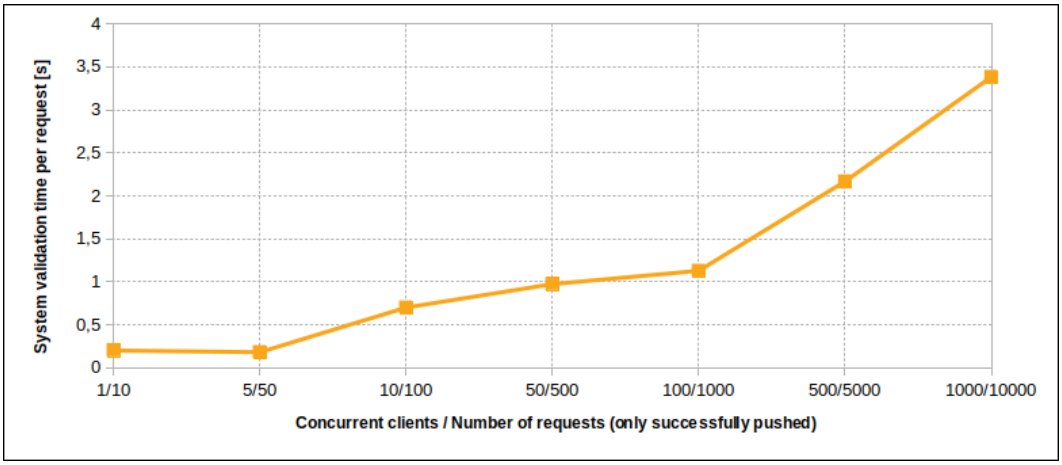

Figure 6.7.: System validation time per request in relation to the concurrent clients and number of requests

7. Conclusion

The main research question of this thesis is:
How can we design a system able to detect smishing SMS effectively and efficiently to protect our customers?

This research question has been aswered here by the proposed design solution and validation. This solution consists of a system called Smishing Detector, which validates the content of an SMS through an API, returning whether the message is legitimate if it does not contain smishing, or illegitimate if it does. This system accepts multiple languages within the SMS, in its initial version English and Spanish. This system is easily scalable and therefore in the future support for more languages could be added, for example Portuguese or French, among many others.

The design and implementation of the system has been evaluated, tested and optimized to ensure the highest possible performance and effectiveness in the context of SMS validation. Likewise, the system has been designed to be easily integrated, guaranteeing its rapid use and providing support in possible cases of error.

During the results phase, it has been observed that the system works better when the content of the messages is in Spanish, compared to messages in English. One of the key factors for the degree of accuracy of our system is the dataset used for the training of each validation model. It has been observed that the system detects legitimate messages perfectly, but when the messages are illegitimate in some cases they end up being detected as legitimate. Therefore, to improve this point we could use new prediction techniques and improve our training datasets.

Another fundamental aspect is the speed of the system and the capacity to take on large workloads. Lleida.net is a company with a high volume of SMS per second, and therefore the system needs a high capacity to assume a greater number of tasks. It has been observed that the absorption of requests by the API is high, but limited by the capabilities of the server where the system has been deployed. In order to increase the number of total requests and concurrent clients we could increase the performance of the server, such as the number of cores or ram memory, and thus accept larger workloads.

Since the system covers a wide number of aspects, it also has its limitations. Likewise, there are many things that have been left out and will be raised as future improvements to our system.

7.1. Limitations

The main limitation of this thesis is with regard to the external URL validation. In order to validate the content of a URL or retrieve information, we must first reach the end of that URL. This implies having to wait until the URL we are trying to reach responds to us and therefore, it is a time that has its limitations. While it is true, when we have to validate multiple URLs we can make a concurrent division of labor, but always depending on the response time of the last case.

This limitation affects our system very directly, since if we want to implement this improvement in real time and quickly we will never be able to estimate the response time since it may contain URLs. A possible solution to this would be to rely solely on prediction models and blacklists to avoid having to check every URL in the message.

Another possible solution would be to implement a kind of cache to store the URLs information and thus avoid having to check them again. This method would only work with the URLs that are inside the cache, so when we talk about a new URL we would have no choice but to visit it and wait for the time needed.

Another clear limitation is the total number of requests that we accept or in other words, the volume of work that our system supports before saturating the resources of the machine. For this limitation we have multiple solutions, one of them would be simply to increase the performance of the machine, that is to say, to increase the number of CPUs, the size of the RAM memory, the number of sockets, etc. Many times increasing the performance of a particular machine is limited by the economic factor and therefore cannot be possible.

Another approach would be to replicate servers with identical validation systems and use a load balancer to manage the amount of traffic that each server receives, thus increasing the effective total number of requests that our overall system can support. As in the previous point, this solution is limited by the economic resources of the company where the system is deployed.

7.2. Future Work

The limitation mentioned in the previous section, has opened interesting opportunities for a future reseach. First, the validation of the URLs, as previously mentioned, another system could be implemented to store the information about these URLs for a period of time acting as a cache memory and avoiding the response time of these URLs.

Regarding the accuracy of the prediction models, since there is still margin for improvement in the metrics of both models, we could improve the training datasets and test different new prediction models. If we can improve the model accuracy, we could eliminate the validation of URLs and thus remove one of the limitations of our system, making it possible to use it as a real-time SMS validation service.

Another point of improvement would be the implementation of callbacks, that is to say, a notification by means of which to warn when a validation has reached its end, either correctly or not. With this we will avoid the constant consultation to check if a request has already finished and therefore, we would lighten the workload of the server.

Also, it would be interesting to dockerize our system, so that the installation or deployment of the same one is easier and therefore in case of having different servers to be able to replicate the system in a fast way.

Finally, another aspect to improve would be the dynamic feeding of blacklists. By this we mean that as our system detects illegitimate traffic, all URLs, telephone numbers and emails are automatically inserted into the blacklists. In this way, when we receive traffic with these same values, we avoid having to perform all the validations and guarantee a faster response.

References

[1] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, B. VanderSloot, E. Wustrow, S. Zanella-Béguelin, and P. Zimmermann. Imperfect forward secrecy: How Diffie-Hellman fails in practice. In *22nd ACM Conference on Computer and Communications Security*, Oct. 2015.

[2] T. A. Almeida, J. M. G. Hidalgo, and A. Yamakami. Contributions to the study of sms spam filtering: new collection and results. In *Proceedings of the 11th ACM symposium on Document engineering*, pages 259–262, 2011.

[3] J. Cendrowska. Prism: An algorithm for inducing modular rules. *International Journal of Man-Machine Studies*, 27(4):349–370, 1987.

[4] F. Chiroma, M. Cocea, and H. Liu. Evaluation of rule-based learning and feature selection approaches for classification. *OASIcs*, 2019.

[5] M. Chowdhury. Building a classifier employing prism algorithm with fuzzy logic. *International Journal of Data Mining & Knowledge Management Process (IJDKP) Vol*, 7, 2017.

[6] W. W. Cohen. Fast effective rule induction. In *Machine learning proceedings 1995*, pages 115–123. Elsevier, 1995.

[7] A. K. Jain and B. Gupta. Rule-based framework for detection of smishing messages in mobile environment. *Procedia Computer Science*, 125:617–623, 2018.

[8] S. Krusche and L. Alperowitz. Introduction of continuous delivery in multi-customer project courses. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 335–343, 2014.

[9] S. Mishra and D. Soni. Smishing detector: A security model to detect smishing through sms content analysis and url behavior analysis. *Future Generation Computer Systems*, 108(1):803–815, 2020.

[10] B. Nuseibeh and S. Easterbrook. Requirements engineering: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 35–46, 2000.

[11] R. Panthong and A. Srivihok. Wrapper feature subset selection for dimension reduction based on ensemble learning algorithm. *Procedia Computer Science*, 72:162–169, 2015.

[12] I. Sommerville and P. Sawyer. *RE: a good practice guide*. John Wiley and Sons, 1997.

[13] G. Sonowal and K. Kuppusamy. Smidca: an anti-smishing model with machine learning approach. *The Computer Journal*, 61(8):1143–1157, 2018.

[14] K. Waters. Prioritization using moscow. *Agile Planning*, 12:31, 2009.

[15] K. Wiegers and J. Beatty. *Software requirements*. Pearson Education, 2013.

A. Annex I - Classification Metrics

Accuracy

- https://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html#sklearn.metrics.accuracy_score

The **accuracy** of a test is its ability to differentiate the patient and healthy cases correctly. To estimate the accuracy of a test, we should calculate the proportion of true positive and true negative in all evaluated cases.

The formula for the accuracy is:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{A.1}$$

In multi-label classification, this function computes subset accuracy: the set of labels predicted for a sample must exactly match the corresponding set of labels in y_true.

F1 score, also known as balanced F-score or F-measure

- https://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html#sklearn.metrics.f1_score

The **F1 score** can be interpreted as a weighted average of the precision and recall, where an F1 score reaches its best value at 1 and worst score at 0. The relative contribution of precision and recall to the F1 score are equal.

The formula for the F1 score is:

$$F1 - Score = \frac{2 * (precision * recall)}{(precision + recall)} \tag{A.2}$$

In the multi-class and multi-label case, this is the average of the F1 score of each class with weighting depending on the average parameter.

Precision

- https://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_score.html#sklearn.metrics.precision_score

Precision aims to respond the following question:

- *What proportion of positive predictions was correctly identified?*

The precision is defined as:

$$precision = \frac{TP}{TP + FP} \tag{A.3}$$

where **T**rue **P**ositive (TP) is the number of true positives and **F**alse **P**ositive (FP) the number of false positives. The precision is intuitively the ability of the classifier not to label as positive a sample that is negative.

The best value is 1 and the worst value is 0.

Recall

- https://scikit-learn.org/stable/modules/generated/sklearn.metrics.recall_score.html#sklearn.metrics.recall_score

Recall aims to respond the following question:

- *What proportion of real positive predictions have been identified correctly?*

The recall is defined as:

$$recall = \frac{TP}{TP + FN} \tag{A.4}$$

where TP is the number of true positives and **F**alse **N**egative (FN) the number of false negatives. The recall is intuitively the ability of the classifier to find all the positive samples.

The best value is 1 and the worst value is 0.

B. Annex II - MYSQL creation script

```
##############################################################################################
################################## MySQL Script ##################################
##############################################################################################

CREATE DATABASE IF NOT EXISTS smishing DEFAULT CHARACTER SET latin1;

CREATE TABLE smishing.request (
    id BIGINT NOT NULL COMMENT 'request identifier',
    id_mt int(11) DEFAULT NULL COMMENT 'smsmass mt identifier',
    id_user int(10) NOT NULL COMMENT 'user identifier',
    status smallint(6) NOT NULL DEFAULT '0' COMMENT 'request status',
    status_date int(11) NOT NULL DEFAULT '0' COMMENT 'request status modification timestamp',
    start_date int(11) NOT NULL COMMENT 'request start timestamp ',
    sms_text TEXT NOT NULL COMMENT 'sms text content',
    sms_lang VARCHAR(3) DEFAULT NULL COMMENT 'sms content language in iso 639-1 or iso 639-2 format',
    legitimate SMALLINT(1) DEFAULT NULL COMMENT '1 if sms is legitimate, 0 otherwise',
    PRIMARY KEY (id)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;


CREATE TABLE smishing.email_blacklist (
    value VARCHAR(256) NOT NULL,
    PRIMARY KEY (value)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;


CREATE TABLE smishing.phone_blacklist (
    value VARCHAR(256) NOT NULL,
    PRIMARY KEY (value)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;


CREATE TABLE smishing.url_blacklist (
    value VARCHAR(256) NOT NULL,
    PRIMARY KEY (value)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

C. Annex III - Deployment configurations

C.1. Core

C.1.1. Service file - smishing-core.service

```
[Unit]
Description=Smishing Core

[Service]
User=root
Group=root
WorkingDirectory=/opt/smishing/system
Environment="PATH=/opt/smishing/system/venv/bin"
Restart=always
StandardError=syslog
StandardOutput=syslog
SyslogIdentifier=smishing-core
ExecStart=/opt/smishing/system/smishing.sh
ExecReload=/bin/kill -HUP $MAINPID

[Install]
WantedBy=multi-user.target
```

C.1.2. Shell start script - smishing.sh

```
#!/bin/bash

# i>&j - Redirects file descriptor i to j.
# 3>&1 - moves file descriptor 1 (aka stdout) to file descriptor 3.
# 1>&2 - moves file descriptor 2 (aka stderr) to file descriptor 1.
# 2>&3 - moves file descriptor 3 to file descriptor 2 (aka stderr).
while true
do
    ((./smishing.py 3>&1 1>&2 2>&3) | /bin/tee -a /opt/localstore/logs/smishing.debug) >> /opt/localstore/logs/x.smishing.debug 2>&1
    /bin/sleep 10
done
```

C.1.3. Python start script - smishing.py

```
from core.Engine import Engine
from modules.Logger import Logger
from modules.module import LOG_PATH, line

from os import getpid, kill, remove
from signal import signal, SIGINT, SIGTERM
from sys import exit
from time import sleep

# Handle signals and finalize core
def handler(signal_received, frame):
    logger.info(f"msg: STOPPING CORE! SIGINT or SIGTERM detected!", line())
    sleep(2) # let things finalize
    remove("/var/run/smishing-core.pid")
    exit(0)

# Initialize log
logger = Logger(LOG_PATH, "smishing.py")

# Tell Python to run the handler() function when SIGINT or SIGTERM is recieved
signal(SIGINT, handler)
signal(SIGTERM, handler)

# Get the process ID of the current process
pid = getpid()

try:
    with open("/var/run/smishing-core.pid", "w") as f:
        f.write(f"{pid}\n")
    logger.info(f"msg: STARTING CORE!", line())
    engine = Engine()
    engine.wait_requests() # blocking operation

except Exception as e:
    logger.error(f"msg: an exception ocurred: {e}, goaway!", line())
    kill(pid, SIGTERM)
```

C.2. Gunicorn

C.2.1. Service file - smishing-api.service

```
[Unit]
Description=Gunicorn instance to serve smishing project
After=network.target

[Service]
User=root
Group=root
```

```
WorkingDirectory=/opt/smishing/system
Environment="PATH=/opt/smishing/system/venv/bin"
ExecStart=/opt/smishing/system/venv/bin/gunicorn --workers 3 --bind unix:/var/run/smishing.sock -m 007 api.wsgi:app

[Install]
WantedBy=multi-user.target
```

C.3. NGINX

C.3.1. Service file - nginx.service

```
[Unit]
Description=nginx - high performance web server
Documentation=http://nginx.org/en/docs/
After=network-online.target remote-fs.target nss-lookup.target
Wants=network-online.target

[Service]
Type=forking
PIDFile=/var/run/nginx.pid
ExecStart=/usr/sbin/nginx -c /etc/nginx/nginx.conf
ExecReload=/bin/sh -c "/bin/kill -s HUP $(/bin/cat /var/run/nginx.pid)"
ExecStop=/bin/sh -c "/bin/kill -s TERM $(/bin/cat /var/run/nginx.pid)"

[Install]
WantedBy=multi-user.target
```

C.3.2. NGINX configuration file

```
##########################################################################################################
# NGINX configuration #
##########################################################################################################

# Defines user and group credentials used by worker processes.
# If group is omitted, a group whose name equals that of user is used.
user root;

# The best number of workers to configure is equal to the number of core's of the system.
worker_processes 2;

# Reconfigure the error.log location.
error_log /opt/localstore/logs/error.log;

```

```
# Reconfigure the Process ID location.
pid /var/run/nginx.pid;

# Events block
events {
    # Sets the maximum number of simultaneous connections that can be opened by a worker process.
    worker_connections 1024;
}

# HTTP block
http {

    # Includes
    include mime.types;
    include proxy.conf;

    # Redefine log format.
    log_format main '$remote_addr - $remote_user [$time_local] $status '
        '"$request" $body_bytes_sent "$http_referer" '
        '"$http_user_agent" "$http_x_forwarded_for"';

    # Reconfigure the access.log location
    access_log /opt/localstore/logs/access.log main;

    # Hide NGINX server version
    server_tokens off;

    # Redirect HTTP to HTTPS
    server {
        # Server fundamentals
        listen 80 default_server;
        listen [::]:80 default_server;
        server_name devel3.lnst.es;

        location / {
            return 308 https://$host$request_uri;
        }
    }

    # HTTPS Reverse Proxy
    server {
        # Server fundamentals
        listen 443 ssl http2;
        listen [::]:443 ssl http2;
        server_name devel3.lnst.es;

        # TLS config
        ssl_certificate /etc/nginx/ssl/certificate.crt;
        ssl_certificate_key /etc/nginx/ssl/private.key;
        ssl_session_cache shared:MozSSL:10m;
```

```
65          ssl_session_timeout 1d;
66          ssl_session_tickets off;
67          ssl_protocols TLSv1.2 TLSv1.3;
68
69          # Intermediate configuration
70          ssl_ciphers ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:↩
                ↪ ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-RSA-AES256-GCM-SHA384:ECDHE-↩
                ↪ ECDSA-CHACHA20-POLY1305:ECDHE-RSA-CHACHA20-POLY1305:DHE-RSA-↩
                ↪ AES128-GCM-SHA256:DHE-RSA-AES256-GCM-SHA384;
71          ssl_prefer_server_ciphers off;
72
73          # These parameters define how OpenSSL performs the Diffie-Hellman (DH) ↩
                ↪ key-exchange.
74          ssl_dhparam /etc/nginx/ssl/ffdhe2048.txt;
75
76          # Headers Hardening
77          add_header X-XSS-Protection "1; mode=block";
78          add_header Strict-Transport-Security "max-age=63072000" always;
79          add_header X-Frame-Options DENY;
80          add_header X-Content-Type-Options nosniff;
81
82          # Pass requests to the UNIX socket
83          location / {
84              limit_except GET POST {
85                  deny all;
86              }
87              error_page 403 = @405;
88              proxy_pass http://unix:/var/run/smishing.sock;
89          }
90      }
91  }
```